

JAVA – Le Collection

Metodi Avanzati di Programmazione
Laurea Triennale in Informatica
Università degli Studi di Bari Aldo Moro
Docente: Pierpaolo Basile

Collection

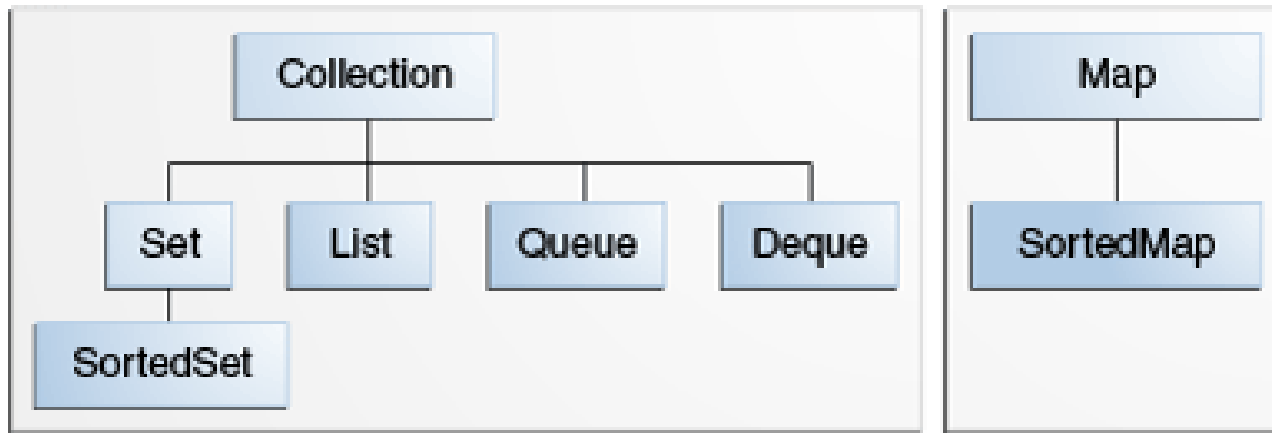
- Una collection è un oggetto che racchiude (contenitore) più oggetti
- E' utilizzata per memorizzare, recuperare ed elaborare gruppi di oggetti
- Rappresenta un gruppo di cose che vanno tenute insieme
 - Un mazzo di carte
 - L'elenco di una rubrica telefonica
 - L'insieme di mail ricevute
 - ...

Il Framework Collection

- JAVA mette a disposizione un framework per la gestione delle Collection composto da
 - **Interfacce**: sono l'astrazione di ogni tipologia di collection
 - **Implementazione**: sono le classi che implementano le interfacce
 - **Algoritmi**: una serie di algoritmi (ricerca, ordinamento, ...) in grado di funzionare in modo *polimorfo* sulle interfacce

Le interfacce di Collection

- Le interfacce permettono di lavorare in maniera indipendente dalla loro effettiva implementazione



*Gerarchia delle interfacce nel framework Collection
(Map è una collection particolare che non eredita da collection)*

Le interfacce di Collection

- Tutte le interfacce Collection sono generiche
`public interface Collection<E>...`
- `<E>` sta ad indicare che la collezione è generica e conterrà solo oggetti di tipo E
- Quanto si crea un'istanza di una classe che implementa Collection bisogna specificare il tipo E (ad es. String, Integer, Double, o un proprio tipo di classe)

Le interfacce di Collection

- **Collection:** è la radice della gerarchia e per questo la più generica, rappresenta semplicemente un contenitore di oggetti (*elementi*) senza alcun particolare vincolo
- **Set:** rappresenta un contenitore di tipo insieme, non può contenere duplicati
- **List:** una lista di elementi, ogni elemento avrà una posizione nella lista, ammette duplicati

Le interfacce di Collection

- **Queue:** è una coda in cui gli elementi hanno un preciso ordine di inserimento e recupero
 - FIFO (first-in-first-out)
 - Ci sono code particolari dette con priorità in cui l'ordine è dettato da una funzione di ordinamento sugli elementi
- **Deque:** simile ad una coda ma permette l'accesso ad entrambe l'estremità della coda

Le interfacce di Collection

- **Map:** permette di collegare dei valori a delle chiavi
 - le chiavi non possono essere duplicate all'interno della stessa Map
- **SortedSet:** un Set in cui gli elementi sono ordinati in ordine crescente
- **SortedMap:** una Map in cui le chiavi sono ordinate in ordine crescente

L'interfaccia Collection

- Tutte le interfacce che ereditano da Collection ereditano i suoi metodi primitivi per gestire un gruppo di oggetti
 - *int size()*: restituisce il numero di oggetti
 - *boolean isEmpty()*: true se è vuota
 - *boolean contains(Object element)*: true se la collection contiene *element*
 - *boolean add(E element)*: aggiunge un elemento
 - *boolean remove(Object element)*: rimuove un elemento
 - *Iterator<E> iterator()*: restituisce un oggetto Iterator che permette di iterare su tutti gli elementi

L'interfaccia Collection

- Metodi che agiscono sull'intera Collection
 - *boolean containsAll(Collection<?> c)*: restituisce true se la collection contiene tutti gli elementi in *c*
 - *boolean addAll(Collection<? extends E> c)*: aggiunge tutti gli elementi in *c* alla collection
 - *boolean removeAll(Collection<?> c)*: rimuove tutti gli elementi di *c* dalla collection
 - *boolean retainAll(Collection<?> c)*: mantiene nella collection solo gli elementi presenti in *c*
 - *void clear()*: elimina tutti gli elementi dalla collection

Iterare su una Collection

Metodo for-each

```
for (Object o : collection)
    System.out.println(o);
```

Metodo iterator

```
Iterator<?> it=collection.iterator()
while (it.hasNext())
    System.out.println(it.next());
```

L'interfaccia iterator

```
public interface  
Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

- hasNext(): restituisce true se ci sono altri elementi da visionare
- next(): restituisce il prossimo elemento
- remove(): rimuove l'elemento corrente

Filtrare elementi da una Collezione

```
Iterator<?> it=collection.iterator()  
while (it.hasNext()) {  
    if (!cond(it.next()))  
        it.remove();  
}
```

È un metodo «fittizio» che decide se un elemento rispetta o meno una particolare condizione

Nel caso sia necessario rimuovere degli elementi è preferibile utilizzare un Iterator e non il metodo for-each

Il metodo toArray

- L'interfaccia Collection ha due metodi toArray che permettono di fare da ponte tra le collection e gli array
 - *Object[] a = c.toArray()*: trasforma la collection *c* in un array di oggetti, *a* avrà la stessa dimensione di *c*
 - *String[] a = c.toArray(new String[0])*: se conosciamo il tipo di elementi in *c* possiamo creare un array che contiene gli stessi elementi di *c* e dello stesso tipo

Un contenitore di tipo insieme che non può contenere duplicati

SET

Set

- Implementa tutti i metodi dell'interfaccia Collection e non ammette duplicati
- Esistono tre implementazioni di Set
 - **HashSet**: un set implementato da una tabella hash non mantiene l'ordine di inserimento degli elementi; è l'implementazione più efficiente
 - **TreeSet**: un set implementato con una struttura ad albero che mantiene l'ordine di inserimento, è meno efficiente
 - **LinkedHashSet**: un set implementato con una tabella hash e puntatore che mantiene l'ordine di inserimento degli elementi

Set

- L'uguaglianza degli oggetti è definita dai metodi *equals* e *hashCode* della classe `Object`
 - la vostra classe può fare l'override di questi metodi definendo il suo concetto di uguaglianza
- Possiamo creare un `Set` a partire dagli elementi di una `Collection c`
 - `Set<Type> s=new LinkedHashSet<Type>(c);`
 - `s` non conterrà duplicati, questo è un modo semplice per rimuovere duplicati da una collection

Set (esempio)

```
public class Esempio1 {  
    public static void main(String[] args) {  
        Set<String> set = new LinkedHashSet<>();  
        set.add("a");  
        set.add("a");  
        set.add("b");  
        set.add("c");  
        System.out.println(set.size() + ": " + set);  
        set.remove("a");  
        System.out.println(set.size() + ": " + set);  
        Set<String> set1 = new LinkedHashSet<>();  
        set1.add("b");  
        set1.add("c");  
        set.removeAll(set1);  
        System.out.println(set.size() + ": " + set);  
    }  
}
```

Operazioni sugli insiemi

```
Set<Type> union = new HashSet<Type>(s1);  
union.addAll(s2);
```

```
Set<Type> intersection = new HashSet<Type>(s1);  
intersection.retainAll(s2);
```

```
Set<Type> difference = new HashSet<Type>(s1);  
difference.removeAll(s2);
```

Una lista di elementi, ogni elemento avrà una posizione nella lista,
ammette duplicati

LIST

List

- Una List è una sequenza di elementi, quindi ogni elemento avrà una sua posizione
- Sono ammessi duplicati
- Oltre ai metodi previsti da Collection ci sono metodi per
 - accedere agli elementi tramite la loro posizione
 - cercare un oggetto e ottenere la posizione in cui si trova
 - iterare sulla sequenza
 - creare delle sottoliste, avere una visione parziale della lista

Implementazioni di List

- `ArrayList()`: generalmente l'implementazione più utilizzata e performante
- `LinkedList()`: implementazione con doppi puntatori

List

- Metodi che sfruttano l'accesso alla posizione
 - `get(int i)`: restituisce l'oggetto alla posizione *i* (le posizioni partono da 0)
 - `set(int i, E element)`: inserisce l'elemento alla posizione *i*
 - `remove(int i)`: elimina l'oggetto in posizione *i*
 - `add(int i, E element)`: aggiunge l'elemento in posizione *i*
 - `addAll(int i, Collection c)`: aggiunge tutti gli elementi in *c* a partire dalla posizione *i*

List

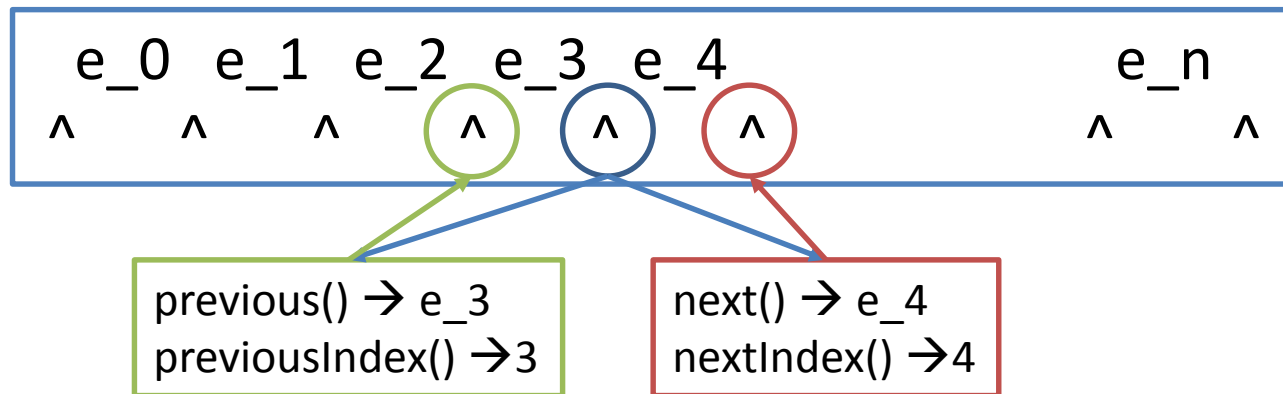
- Metodi per la ricerca degli elementi
 - `indexOf(Object o)`: restituisce la posizione in cui si trova l'oggetto `o`, altrimenti -1
 - `lastIndexOf(Object o)`: restituisce l'ultima posizione in cui si trova l'oggetto `o`, altrimenti -1
 - `o` potrebbe trovarsi in più posizioni nella lista, se duplicato, questo metodo restituisce l'ultima posizione

List (iterazione)

- `iterator()`: restituisce un iteratore su questa lista, gli elementi verranno elencati in sequenza
- Ci sono due metodi che restituiscono un `ListIterator` che permette di scorrere la lista sia in avanti che indietro
 - `listIterator()`: restituisce un `ListIterator` che parte dall'inizio della lista
 - `listIterator(int i)`: restituisce un `ListIterator` che parte dalla posizione i

ListIterator

- Il ListIterator è un iteratore che ammette un cursore tra un elemento e l'altro della lista e permette di andare all'elemento precedente `previous()` e all'elemento successivo `next()`



`previous()` e `next()` spostano il cursore indietro e avanti!!!

ListIterator (altri metodi)

- hasNext(): true se c'è un elemento avanti
- hasPrevious(): true se c'è un elemento indietro
- remove(): rimuove l'elemento ottenuto dall'ultima chiamata di next() o previous()
- add(E e): aggiunge *e* tra previous() e next()
- set(E e): sostituisce con *e* l'elemento ottenuto dall'ultima chiamata di next() o previous()

List (esempio 1)

```
public class List1 {  
    public static void main(String[] args) {  
        List<String> list=new ArrayList<>();  
        list.add("a");  
        list.add("b");  
        list.add("c");  
        list.add("a");  
        System.out.println(list);  
        list.set(0, "z");  
        System.out.println(list);  
        list.add(3, "d");  
        System.out.println(list);  
        System.out.println(list.get(1));  
        list.remove(2);  
        System.out.println(list);  
        System.out.println(list.indexOf("a"));  
        list.add("a");  
        System.out.println(list.lastIndexOf("a"));  
    }  
}
```

List (esempio 2)

```
public class List2 {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<>();  
        list.add("a");  
        list.add("b");  
        list.add("c");  
        list.add("a");  
        list.add("z");  
        list.add("h");  
        ListIterator<String> lit = list.listIterator();  
        while (lit.hasNext()) {  
            System.out.println(lit.previousIndex()+"\t"+lit.nextIndex());  
            System.out.println(lit.next());  
        }  
    }  
}
```

List (esempio 3)

```
public class List3 {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<>();  
        list.add("a");  
        list.add("b");  
        list.add("c");  
        list.add("a");  
        list.add("z");  
        list.add("h");  
        ListIterator<String> lit = list.listIterator(list.size()  
- 1);  
        while (lit.hasPrevious()) {  
            System.out.println(lit.previous());  
        }  
    }  
}
```

subList(int start, int end)

- Permette di ottenere una sottolista a partire dalla posizione start fino alla posizione end (non inclusa)
`System.out.println(list.subList(1, 3));`
- subList restituisce una nuova lista, non viene fatta nessuna modifica sulla lista di partenza

Algoritmi sulle liste

- La classe **Collections** mette a disposizione dei metodi statici per effettuare degli algoritmi sulle classi che implementano Collection
 - sort: ordina gli elementi nella lista
 - shuffle: permuta in modo casuale gli elementi nella lista
 - reverse: inverte l'ordine degli elementi
 - rotate: ruota gli elementi di una lista
 - swap: scambia due elementi nella lista
 - replaceAll: sostituisce un elemento con un altro valore
 - fill: sostituisce tutti gli elementi con un altro valore
 - copy: copia tra due liste
 - binarySearch: ricerca un elemento nella lista utilizzando la ricerca binaria
 - indexOfSubList, lastIndexOfSubList: cercano una sottolista all'interno di una lista

Esempio

```
public class List4 {  
    public static void main(String[] args) {  
        List<String> l = new ArrayList<>();  
        l.add("pippo"); l.add("topolino");  
l.add("paperino");  
        System.out.println(l);  
        Collections.sort(l);  
        System.out.println(l);  
        Collections.shuffle(l);  
        System.out.println(l);  
        Collections.rotate(l, -1);  
        System.out.println(l);  
    }  
}
```

Una coda in cui gli elementi hanno un preciso ordine di inserimento e recupero

QUEUE

L'interfaccia Queue

- `add(E e)`: aggiunge un elemento alla coda, restituisce `true` se l'elemento è inserito altrimenti genera un'eccezione
- `element()`: restituisce l'elemento in testa alla coda senza rimuoverlo, se la coda è vuota genera un'eccezione
- `offer(E e)`: aggiunge un elemento alla coda
- `peek()`: restituisce l'elemento in testa alla coda senza rimuoverlo, oppure `null` se la coda è vuota
- `poll()`: restituisce e rimuove l'elemento in testa alla coda, `null` se la coda è vuota
- `remove()`: restituisce e rimuove l'elemento in testa alla coda, se la coda è vuota genera un'eccezione

L'interfaccia Queue

I metodi di inserimento, cancellazione e recupero sono duplicati e si differenziano sul loro comportamento (eccezione/restituzione valore) nel caso in cui la coda è vuota o ha raggiunto la **capacità massima**

Operazione	Genera un eccezione	Restituiscono un valore
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Implementazioni di Queue

- LinkedList: implementa una coda FIFO
- PriorityQueue: implementa una lista con priorità il cui elemento in cima è sempre quello più piccolo

Queue (esempio 1)

```
public class Queue1 {  
    public static void main(String[] args) {  
        Queue<String> q=new LinkedList<>();  
        q.offer("g");  
        q.offer("h");  
        q.offer("a");  
        System.out.println(q.peek()); //g  
    }  
}
```

Queue (esempio 2)

```
public class Queue2 {  
    public static void main(String[] args) {  
        Queue<String> q=new PriorityQueue<>();  
        q.offer("g");  
        q.offer("h");  
        q.offer("a");  
        System.out.println(q.peek()); //a  
    }  
}
```

Simile ad una coda ma permette l'accesso ad entrambe l'estremità della coda

DEQUE (*DECK*)

Deque

Operazione	Coda	Testa
Insert	addFirst(e) offerFirst(e)	addLast(e) offerLast(e)
Remove	removeFirst() pollFirst()	removeLast() pollLast()
Examine	getFirst() peekFirst()	getLast() peekLast()

ArrayDeque e LinkedList sono due implementazioni di Deque

Deque (esempio)

```
public class Deque1 {  
    public static void main(String[] args) {  
        Deque<String> q = new LinkedList<>();  
        q.offerLast("g");  
        q.offerLast("h");  
        q.offerFirst("z");  
        q.offerLast("a");  
        System.out.println("Coda: " + q.peekLast());  
        System.out.println("Testa: " + q.peekFirst());  
    }  
}
```

Permette di collegare dei valori a delle chiavi

MAP

Map

- Map permette di creare delle associazioni chiave-valore
 - le chiavi non possono essere duplicate
 - ad ogni chiave è associato un solo valore
 - è l'astrazione del concetto di *funzione*
- Esistono tre implementazioni (in maniera analoga a set)
 - HashMap
 - TreeMap
 - LinkedHashMap

Map (operazioni base)

- `get(Object key)`: restituisce l'oggetto associato alla chiave *key*
- `put(K key, V value)`: inserisce una coppia chiave-valore nella Map
- `remove(Object key)`: rimuove la coppia chiave-valore associata a *key*
- `containsKey(Object key)`, : restituisce true se la Map contiene la chiave *key*
- `containsValue(Object value)`: restituisce true se la Map contiene il valore *value*
- `putAll(Map<K, V> map)`: inserisce tutti gli elementi di *map*

Map (iterazione)

- `keySet`: restituisce un Set contenente tutte le chiavi
- `values()`: restituisce una collection che contiene tutti i valori
- `entrySet()`: restituisce un Set di tutte le coppie chiave-valore

Map (esempio 1)

```
public class Freq {  
    public static void main(String[] args) {  
        Map<String, Integer> m = new HashMap<>();  
  
        // Initialize frequency table from command line  
        for (String a : args) {  
            Integer freq = m.get(a);  
            m.put(a, (freq == null) ? 1 : freq + 1);  
        }  
  
        System.out.println(m.size() + " distinct words:");  
        System.out.println(m);  
    }  
}
```

Map (es. iterazione)

```
for (KeyType key : m.keySet())  
    System.out.println(key);
```

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())  
    System.out.println(e.getKey() + ": " + e.getValue());
```

```
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext();)   
    if (it.next().isBogus())  
        it.remove();
```


Map (esercizio)

- Leggere tutte le parole presenti nella stringa passata come argomento al main
 - individuare le parole utilizzando i whitespace
- Contare quante volte compare una parola nella stringa
- Stampare a video i risultati del conteggio

Map (soluzione)

```
public class CountWord {  
    public static void main(String[] args) {  
        Map<String, Integer> map = new HashMap<>();  
        Scanner s = new Scanner(args[0]);  
        while (s.hasNext()) {  
            String t = s.next();  
            Integer freq = map.get(t);  
            map.put(t, (freq == null) ? 1 : freq + 1);  
        }  
        System.out.println(map);  
    }  
}
```

Note

EQUALS

Equals

- Ogni classe eredita da Object il metodo equals
- Il metodo equals restituisce true se due oggetti sono uguali
- Ogni classe può fare l'override del metodo equals in base al concetto di uguaglianza della classe
- Il metodo equals è utilizzato da Set per evitare duplicati, ma anche da tutti i metodi di Collection che cercano un oggetto

hashCode

- Ogni classe eredita da Object il metodo hashCode() che restituisce un int
- Il metodo restituisce il codice hash per l'oggetto
 - è utilizzato dalle tabelle hash come quelle fornite da HashMap
- Se due oggetti sono uguali in base al metodo equals (Object), la chiamata del metodo hashCode su ciascuno dei due oggetti deve produrre **gli stessi** risultati interi
 - Object genera l'hash code utilizzando l'indirizzo di memoria dell'oggetto

Equals (esempio)

```
public class Person {  
    private String cf;  
    private String name;  
    private String surname;  
    private int age;  
  
    public Person(String cf, String name, String surname, int age) {  
        this.cf = cf;  
        this.name = name;  
        this.surname = surname;  
        this.age = age;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (obj!=null && obj instanceof Person) {  
            Person p = (Person) obj;  
            return p.cf.equals(this.cf);  
        } else {  
            return false;  
        }  
    }  
}
```

Due persone sono uguali se hanno lo stesso *cf*

ORDINAMENTO

Ordinamento

- Come per il metodo equals ogni classe può avere una sua definizione di ordinamento
 - ad esempio per la classe Person l'ordinamento potrebbe essere in base all'età (*age*)
- Per definire un ordinamento sugli oggetti di una classe la classe deve implementare l'interfaccia **Comparable**

L'interfaccia Comparable


- Prevede solo un metodo: `compareTo(T o)`
 - restituisce 0 se gli oggetti sono uguali
 - -1 se l'oggetto è minore di `o`
 - +1 se l'oggetto è maggiore di `o`

Comparable (esempio)

```
public class Person implements Comparable<Person> {
```

```
    //codice della classe Person
```

```
    public int compareTo(Person o) {  
        return Integer.compare(this.age, o.age);  
    }
```



In questo caso sfruttiamo il metodo compare di Integer per confrontare due interi

```
}
```

L'interfaccia Comparator

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Il metodo compare funziona in maniera analoga al compareTo

- 0 se o1 è uguale a o2
- <0 se o1 è minore di o2
- >0 se o1 è maggiore di o2

Esiste un metodo sort di Collection che ha come argomento una lista e un oggetto di tipo Comparator. Il Comparator è utilizzato come funzione di ordinamento per ordinare gli oggetti nella lista

Un Set in cui gli elementi sono ordinati

SORTED SET

SortedSet

```
public interface SortedSet<E> extends Set<E>
{
    // Range-view
    SortedSet<E> subSet(E fromElement, E
toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Gli elementi sono ordinati in modo decrescente in base al loro ordinamento naturale o può essere passato un Comparator nel costruttore

Le range-view permettono viste sugli oggetti sfruttando il loro ordinamento

Il primo e l'ultimo elemento del set

Permette di accedere al Comparator di questo set

SortedSet (esempio)

```
public class SortSet1 {  
  
    public static void main(String[] args) {  
        SortedSet<String> s=new TreeSet<>();  
        s.add("a"); s.add("b"); s.add("c"); s.add("l");  
s.add("m"); s.add("y");  
        System.out.println(s.subSet("d", "z"));  
        System.out.println(s.headSet("l"));  
        System.out.println(s.tailSet("h"));  
        System.out.println(s.first());  
        System.out.println(s.last());  
    }  
  
}
```

Mantiene le coppie chiave-valore ordinate in base alla chiave

SORTEDMAP

SortedMap

```
public interface SortedMap<K, V>  
    extends Map<K, V>{  
        Comparator<? super K>  
        comparator();
```

```
        SortedMap<K, V> subMap(K  
        fromKey, K toKey);  
        SortedMap<K, V> headMap(K  
        toKey);  
        SortedMap<K, V> tailMap(K  
        fromKey);
```

```
        K firstKey();  
        K lastKey();  
    }
```

Gli elementi sono ordinati in modo decrescente in base all'ordinamento naturale delle chiavi o può essere passato un Comparator nel costruttore

Le range-view permettono viste sugli oggetti sfruttando l'ordinamento sulle chiavi

La prima e l'ultima chiave

GENERIC

Generics

- Le Collection sono strutture generiche che memorizzano oggetti di tipo Object
- Abbiamo visto che è possibile tipizzare le Collection utilizzando i generics <T>
 - in questo modo tutti metodi invece che lavorare su oggetti di tipo Object lavoreranno su oggetti di tipo T
- I generics sono uno strumento potente per tipizzare il comportamento delle classi

Generics

Senza generics

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

E' necessaria l'operazione di cast (cambio del tipo di oggetto)

Con generics

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0); // no cast
```

Il metodo get è stato tipizzato in base al generics <String>

Generics nella definizione di una classe

```
public class Box {  
    private Object object;  
  
    public void set(Object object) {  
this.object = object; }  
  
    public Object get() { return object;  
}  
  
}
```

```
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

Uso della classe Box

```
Box<String> stringBox=new Box<>();  
stringBox.set("pippo");
```

```
Box<Integer> intBox=new Box<>();  
intBox.set(42);
```

Esercizio 1

- Progettare la gestione di un negozio online considerando che:
 - Ogni utente ha un id, username, uno storico degli ordini effettuati
 - Ci sono due tipologie di utenti: utente normale che paga il costo di spedizione normale e un utente «prime» che paga un costo di spedizione fisso di 1€ ad ordine
 - Ogni articolo è caratterizzato da un id, una descrizione, un costo unitario, un peso in grammi
 - Ogni ordine è caratterizzato dall'utente che ha effettuato l'ordine e dalla lista di articoli inseriti nell'ordine e dal costo totale (diviso in costo merce e costo spedizioni)
 - Il magazzino raccoglie tutti gli articoli disponibili e per ogni articolo la quantità presente nel magazzino
 - Le spese di spedizione di un ordine dipendono dal peso e sono di 2€ per ogni chilogrammo, se il peso è minore di un chilogrammo il costo è 2€ (gli utenti «prime» pagano 1€ per ogni ordine)

Esercizio 1

- Realizzare la seguente funzione:
 - Simulare il carrello di un potenziale acquisto, se il magazzino ha a disposizione gli oggetti presenti nel carrello
 - Calcolare il costo complessivo del carrello
 - Calcolare le spese di spedizione
 - Assegnare all'utente l'ordine inserendolo nello storico degli ordini dell'utente

Esercizio 1

- Realizzare le seguenti funzioni:
 - Ricerca di un articolo nel magazzino
 - Visualizzazione di tutti gli ordini di un utente
 - Ricerca degli articoli nel magazzino la cui disponibilità è inferiore ad una certa quantità
 - Lista dei primi 5 articoli più venduti

THE END

