9. Il Java Collections Framework

Questo capitolo è dedicato all'esplorazione del **Java Collections Framework (JCF)**, una componente fondamentale della libreria standard di Java per la gestione di gruppi di oggetti. Il JCF fornisce un'**architettura unificata** per **rappresentare e manipolare collezioni**, offrendo interfacce, implementazioni e algoritmi.

3.1. Introduzione al Java Collections Framework (JCF)

Il JCF è una **libreria** del linguaggio Java, parte del package `java.util`, progettata per **gestire strutture dati (o collezioni)** e i relativi algoritmi. La sua importanza è duplice:

- **Importanza Pratica**: Virtualmente ogni sistema software fa uso di collezioni di oggetti. Conoscere la struttura e i dettagli del JCF è essenziale per scrivere codice efficiente e robusto.
- **Importanza Didattica**: Il JCF fornisce eccellenti esempi di utilizzo di composizione, ereditarietà e genericità, mettendo in pratica pattern di programmazione di interesse e impattando vari aspetti del linguaggio.

3.1.1. Vantaggi del JCF

L'utilizzo del JCF porta a diversi benefici:

- **Riduzione del Lavoro di Programmazione**: Non è necessario scrivere le proprie implementazioni di strutture dati comuni.
- **Aumento delle Prestazioni**: Le implementazioni del JCF sono ottimizzate per le prestazioni.
- **Interoperabilità tra API non Correlate**: Permette a diverse API di scambiare collezioni di oggetti.
- **Facilità di Apprendimento e Utilizzo**: Fornisce un'API coerente e intuitiva.
- **Riduzione dell'Effort per Progettare Nuove API**: Le nuove API possono basarsi su interfacce e implementazioni standard.

3.2. Gerarchia delle Interfacce del JCF

Il JCF è basato su una **gerarchia di interfacce** che definiscono i **tipi di collezione** e i loro comportamenti. Le interfacce principali sono `Collection`, `Set`, `List`, `Queue` e `Map`.

3.2.1. Interfaccia `Collection<E>`

`Collection<E>` è la radice della gerarchia delle collezioni. Rappresenta un gruppo di elementi. Non è direttamente implementata, ma serve come base per interfacce più specifiche.

Metodi principali:

- `boolean add(E e)`: Aggiunge un elemento.
- `boolean remove(Object o)`: Rimuove un elemento.
- `boolean contains(Object o)`: Verifica la presenza di un elemento.
- `int size()`: Restituisce il numero di elementi.
- `boolean isEmpty()`: Verifica se la collezione è vuota.
- `Iterator<E> iterator()`: Restituisce un iteratore per la collezione.
- `Object[] toArray()`: Restituisce un array contenente tutti gli elementi.
- `boolean addAll(Collection<? extends E> c)`: Aggiunge tutti gli elementi di un'altra collezione.
- `boolean removeAll(Collection<?> c)`: Rimuove tutti gli elementi contenuti anche in un'altra collezione.
- `boolean retainAll(Collection<?> c)`: Mantiene solo gli elementi contenuti anche in un'altra collezione.
- `void clear()`: Rimuove tutti gli elementi.

```
import java.util.Iterator;

interface Collection<E> {
```

```
    boolean add(E e);
    boolean remove(Object o);
    boolean contains(Object o);
    int size();
    boolean isEmpty();
    Iterator<E> iterator();
    Object[] toArray();
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
}
```

3.2.2. Interfaccia `Set<E>`

`Set<E>` estende `Collection<E>` e rappresenta una collezione che **non contiene elementi duplicati**. Modella il concetto matematico di **insieme**.

Caratteristiche:

- **Unicità**: Ogni elemento può apparire al massimo una volta.
- **Ordine**: L'ordine degli elementi non è garantito (a meno che non si usi un'implementazione specifica come `LinkedHashSet` o `TreeSet`).

Implementazioni comuni:

- `HashSet<E>`: Implementazione basata su tabella hash. Offre prestazioni medie a tempo costante (O(1)) per le operazioni di base (add, remove, contains), ma l'ordine di iterazione non è garantito e può cambiare.
- `LinkedHashSet<E>`: Estende `HashSet`, **mantiene l'ordine** di inserimento degli elementi.
- `TreeSet<E>`: Implementazione basata su **albero binario di ricerca** (Red-Black Tree). Mantiene gli **elementi ordinati** (naturalmente o tramite un `Comparator`). Le operazioni di base hanno complessità logaritmica (O(logn)).

```
import java.util.Collection;
import java.util.Iterator;

public interface Set<E> extends Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean retainAll(Collection<?> c);
    boolean removeAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
}
```

3.2.3. Interfaccia `List<E>`

`List<E>` estende `Collection<E>` e rappresenta una collezione **ordinata** di elementi, che **può contenere duplicati**. Permette l'accesso agli elementi tramite indice (posizione).

Caratteristiche:

- **Ordine**: Gli elementi hanno un ordine definito e sono accessibili tramite indice.
- **Duplicati**: Può contenere più occorrenze dello stesso elemento.

Metodi aggiuntivi rispetto a `Collection`:

- `void add(int index, E element)`: Inserisce un elemento in una posizione specifica.
- `E get(int index)`: Restituisce l'elemento in una posizione specifica.
- `E set(int index, E element)`: Sostituisce l'elemento in una posizione specifica.
- `E remove(int index)`: Rimuove l'elemento in una posizione specifica.
- `int indexOf(Object o)`: Restituisce l'indice della prima occorrenza di un elemento.
- `int lastIndexOf(Object o)`: Restituisce l'indice dell'ultima occorrenza di un elemento.
- `ListIterator<E> listIterator()`: Restituisce un iteratore specifico per le liste.

Implementazioni comuni:

- `ArrayList<E>`: Implementazione basata su un **array ridimensionabile**. Ottima per l'accesso casuale (O(1) per `get(index)`) e l'aggiunta in coda. L'inserimento o la rimozione in posizioni arbitrarie sono costosi (O(n)).
- `LinkedList<E>`: Implementazione basata su una **lista doppiamente collegata**. Ottima per l'inserimento e la rimozione in qualsiasi posizione (O(1) se si ha già il riferimento al nodo), ma l'accesso casuale è costoso (O(n)).
- `Vector<E>`: Simile a `ArrayList`, ma sincronizzata (thread-safe). Generalmente sconsigliata per le prestazioni, a favore di `ArrayList` e meccanismi di sincronizzazione esterni se necessari.

3.2.4. Interfaccia `Queue<E>`

`Queue<E>` estende `Collection<E>` e rappresenta una collezione progettata per **contenere elementi prima dell'elaborazione**. Le code seguono tipicamente un ordine FIFO (First-In, First-Out).

Metodi principali (operazioni su testa e coda):

- `boolean add(E e)` / `boolean offer(E e)`: Aggiunge un elemento alla coda. `add` lancia un'eccezione se la coda è piena, `offer` restituisce `false`.
- `E remove()` / `E poll()`: Rimuove e restituisce l'elemento in testa. `remove` lancia un'eccezione se la coda è vuota, `poll` restituisce `null`.
- `E element()` / `E peek()`: Restituisce l'elemento in testa senza rimuoverlo. `element` lancia un'eccezione se la coda è vuota, `peek` restituisce `null`.

Implementazioni comuni:

- `LinkedList<E>`: Può essere usata come implementazione di `Queue`.
- `PriorityQueue<E>`: Una coda che ordina gli elementi in base a un ordine naturale o a un `Comparator`. L'elemento con la priorità più alta (il più piccolo) è sempre in testa.

3.3. Interfaccia `Map<K, V>`

`Map<K, V>` non estende `Collection<E>` ma è parte del JCF. Rappresenta una collezione di **coppie chiave-valore**, dove ogni chiave è unica e mappa a un singolo valore. Modella il concetto matematico di funzione discreta.

Caratteristiche:

- **Coppie Chiave-Valore**: Ogni elemento è una coppia (`Key`, `Value`).
- **Chiavi Uniche**: Ogni chiave può apparire al massimo una volta.
- **Valori Duplicati**: I valori possono essere duplicati.

Metodi principali:

- `V put(K key, V value)`: Associa il valore specificato alla chiave specificata. Se la chiave era già presente, il vecchio valore viene sostituito e restituito.
- `V get(Object key)`: Restituisce il valore associato alla chiave, o `null` se la chiave non è presente.
- `V remove(Object key)`: Rimuove la mappatura per la chiave specificata.
- `boolean containsKey(Object key)`: Verifica se la mappa contiene la chiave specificata.

- `boolean containsValue(Object value)`: Verifica se la mappa contiene il valore specificato.
- `int size()`: Restituisce il numero di mappature.
- `Set<K> keySet()`: Restituisce un `Set` delle chiavi contenute nella mappa.
- `Collection<V> values()`: Restituisce una `Collection` dei valori contenuti nella mappa.
- `Set<Map.Entry<K, V>> entrySet()`: Restituisce un `Set` delle coppie chiave-valore (Entry) contenute nella mappa.

Implementazioni comuni:

- `HashMap<K, V>`: Implementazione basata su tabella hash. Offre prestazioni medie a tempo costante (O(1)) per le operazioni di base (put, get, remove), a discapito di un overhead in memoria. L'ordine di iterazione non è garantito.
- `LinkedHashMap<K, V>`: Estende `HashMap`, **mantiene l'ordine di inserimento** delle coppie chiave-valore.
- `TreeMap<K, V>`: Implementazione basata su **albero binario di ricerca** (Red-Black Tree). Mantiene le chiavi **ordinate** (naturalmente o tramite un `Comparator`). Le operazioni di base hanno complessità logaritmica (O(logn)).

Esempio di utilizzo di `Map`:

```
import java.util.HashMap;
import java.util.Map;

public class UseMap {
   public static void main(String[] args) {
      // Uso una incarnazione, ma poi lavoro sull'interfaccia
      final Map<Integer, String> map = new HashMap<>();
      // Una mappa è una funzione discreta
      map.put(345211, "Bianchi");
      map.put(345122, "Rossi");
      map.put(243001, "Verdi");
      System.out.println(map); // {345211=Bianchi, 243001=Verdi, 345122=Rossi}
      map.put(243001, "Neri"); // Rimpiazza Verdi
      System.out.println(map); // {345211=Bianchi, 243001=Neri, 345122=Rossi}

      final Map<String,Integer> map2 = Map.of("foo", 5, "bar", 7); // Java 9+ per Map.of()
      // Modo prestante per accedere alle coppie
      for(final Map.Entry<String,Integer> kv : map2.entrySet()) {
         System.out.println("Chiave: " + kv.getKey() + ", Valore: " + kv.getValue());
      }
   }
}
```

3.4. Iteratori e `for-each` Loop

Il JCF fornisce meccanismi standard per attraversare gli elementi di una collezione.

3.4.1. Interfaccia `Iterator<E>`

L'interfaccia `Iterator<E>` fornisce un modo standard per **accedere sequenzialmente** agli elementi di una collezione **senza esporre la sua struttura interna**.

Metodi principali:

- `boolean hasNext()`: Restituisce `true` se l'iterazione ha ancora elementi.
- `E next()`: Restituisce il prossimo elemento nell'iterazione.
- `void remove()`: Rimuove l'ultimo elemento restituito da `next()` dalla collezione sottostante (opzionale).

Esempio di utilizzo di `Iterator`:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
```

```
public class UseIterator {
    public static void main(String[] args) {
        final List<String> list = new ArrayList<>();
        list.add("foo");
        list.add("bar");
        list.add("baz");

        final Iterator<String> it = list.iterator();
        while (it.hasNext()) {
            final String s = it.next();
            System.out.println(s);
        }
    }
}
```

3.4.2. Il `for-each` Loop (Enhanced for Loop)

Il `for-each` loop (introdotto in Java 5) è una **sintassi più concisa e leggibile per iterare** su collezioni e array. Funziona con qualsiasi oggetto che implementi l'interfaccia `Iterable`.

Sintassi: `for (Type element : collectionOrArray) { ... }`

Esempio:

```
import java.util.ArrayList;
import java.util.List;

public class UseForEach {
    public static void main(String[] args) {
        final List<String> list = new ArrayList<>();
        list.add("foo");
        list.add("bar");
        list.add("baz");

        for (final String s : list) { // Uso del for-each loop
            System.out.println(s);
        }
    }
}
```

Internamente, il `for-each` loop viene tradotto dal compilatore in un'iterazione con `Iterator`.

3.4.3. `ListIterator<E>` (per le Liste)

`ListIterator<E>` è un iteratore più potente, specifico per le liste, che consente di:

- Iterare in entrambe le direzioni (avanti e indietro).
- Modificare la lista durante l'iterazione (aggiungere, sostituire, rimuovere elementi).
- Ottenere l'indice corrente dell'elemento.

3.5. Funzioni di Utilità in `Arrays` e `Collections`

Le classi `java.util.Arrays` e `java.util.Collections` forniscono **metodi statici** di utilità per **manipolare array e collezioni**.

3.5.1. Classe `java.util.Arrays`

Fornisce metodi statici per operazioni comuni sugli array:

- `sort()`: Ordina un array.
- `binarySearch()`: Ricerca binaria in un array ordinato.

- `equals()`: Confronta due array per uguaglianza.
- `fill()`: Riempie un array con un valore specifico.
- `copyOf()`: Crea una copia di un array.
- `toString()`: Restituisce una rappresentazione stringa di un array.
- `deepToString()`: Restituisce una rappresentazione stringa ricorsiva per array di array.

3.5.2. Classe `java.util.Collections`

Fornisce metodi statici per operazioni comuni sulle collezioni:

- `sort()`: Ordina una `List`.
- `binarySearch()`: Ricerca binaria in una `List` ordinata.
- `reverse()`: Inverte l'ordine degli elementi in una `List`.
- `shuffle()`: Mescola casualmente gli elementi in una `List`.
- `max()` / `min()`: Restituisce l'elemento massimo/minimo in una `Collection`.
- `frequency()`: Conta le occorrenze di un elemento in una `Collection`.
- `unmodifiableCollection()`, `unmodifiableList()`, `unmodifiableSet()`, `unmodifiableMap()`: Restituiscono viste non modificabili delle collezioni, utili per la programmazione difensiva.

3.6. Considerazioni sulla Scelta delle Implementazioni

La scelta dell'implementazione corretta per una collezione dipende dai requisiti specifici dell'applicazione in termini di prestazioni, ordine degli elementi e unicità.

Interfaccia	Implementazione	Caratteristiche Principali	Complessità Operazioni Base	Note
`Set`	`HashSet`	Non ordinato, no duplicati	O(1) (media)	Veloce, ma ordine non garantito.
	`LinkedHashSet`	Ordine di inserimento, no duplicati	O(1) (media)	Mantiene l'ordine di inserimento.
	`TreeSet`	Ordinato (naturale/Comparator), no duplicati	O(logn)	Elementi sempre ordinati.
`List`	`ArrayList`	Ordinato, duplicati, accesso per indice	`get(index)`: O(1); `add/remove(index)`: O(n)	Ottimo per accesso casuale e aggiunta in coda.
	`LinkedList`	Ordinato, duplicati, basato su nodi	`add/remove(first/last)`: O(1); `get(index)`: O(n)	Ottimo per inserimenti/rimozioni frequenti in testa/coda.
`Map`	`HashMap`	Coppie chiave-valore, chiavi uniche, non ordinato	O(1) (media)	Veloce, ma ordine non garantito.
	`LinkedHashMap`	Ordine di inserimento, chiavi uniche	O(1) (media)	Mantiene l'ordine di inserimento delle coppie.
	`TreeMap`	Chiavi ordinate, chiavi uniche	O(logn)	Chiavi sempre ordinate.

3.7. Collezioni Immutabili (Java 9+)

A partire da Java 9, sono stati introdotti metodi factory per creare collezioni immutabili in modo conciso: `List.of()`, `Set.of()`, `Map.of()`, `Map.ofEntries()`.

Vantaggi delle collezioni immutabili:

- **Thread-Safety**: Possono essere condivise tra più thread senza problemi di sincronizzazione.
- **Sicurezza**: Il loro stato non può essere modificato dopo la creazione.
- **Semplicità**: Riducono la complessità del codice, in particolare in contesti funzionali.

Esempio:

```
import java.util.List;
import java.util.Set;
```

```
import java.util.Map;

public class ImmutableCollections {
    public static void main(String[] args) {
        // Lista immutabile
        List<String> immutableList = List.of("Apple", "Banana", "Cherry");
        System.out.println("Immutable List: " + immutableList);

        // Set immutabile
        Set<Integer> immutableSet = Set.of(1, 2, 3);
        System.out.println("Immutable Set: " + immutableSet);

        // Mappa immutabile
        Map<String, Integer> immutableMap = Map.of("One", 1, "Two", 2, "Three", 3);
        System.out.println("Immutable Map: " + immutableMap);

        // Tentativo di modifica (lancerà UnsupportedOperationException)
        try {
            immutableList.add("Date");
        } catch (UnsupportedOperationException e) {
            System.out.println("Errore: Impossibile modificare una lista immutabile.");
        }
    }
}
```