10
Java Collections Framework

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2025/2026

Outline

Goal della lezione

- Illustrare la struttura del Java Collections Framework
- Mostrare gli utilizzi delle funzonalità base
- Discutere alcune tecniche di programmazione correlate

Argomenti

- Presentazione Java Collections Framework
- Iteratori e foreach
- Collezioni, Liste e Set
- `HashSet` e `TreeSet`

Java Collections Framework

Java Collections Framework (JCF)

- È una libreria del linguaggio Java
- È una parte del package `java.util`
- Gestisce strutture dati (o collezioni) e relativi algoritmi

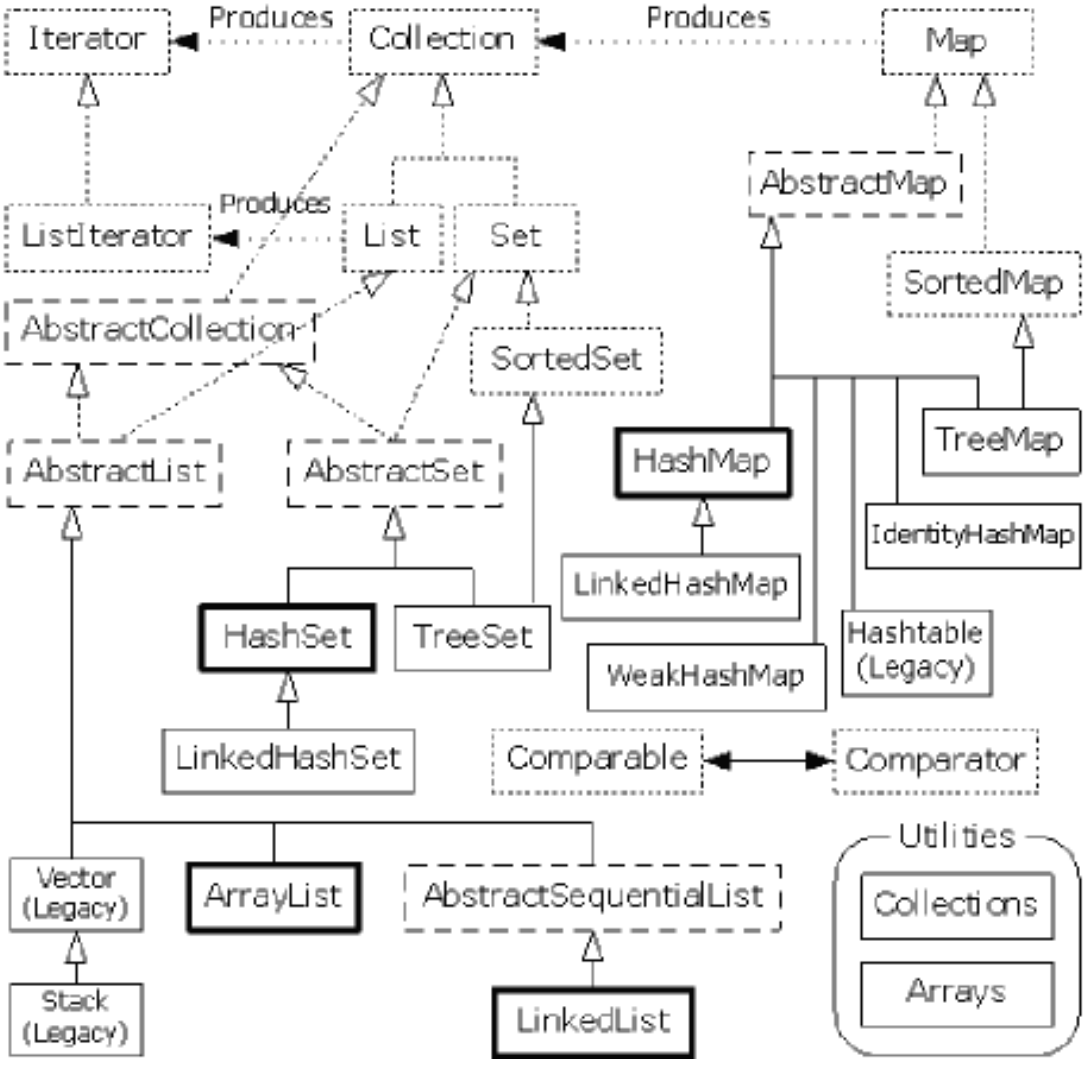
Importanza pratica

- Virtualmente ogni sistema fa uso di collezioni di oggetti
- Conoscerne struttura e dettagli vi farà programmatori migliori

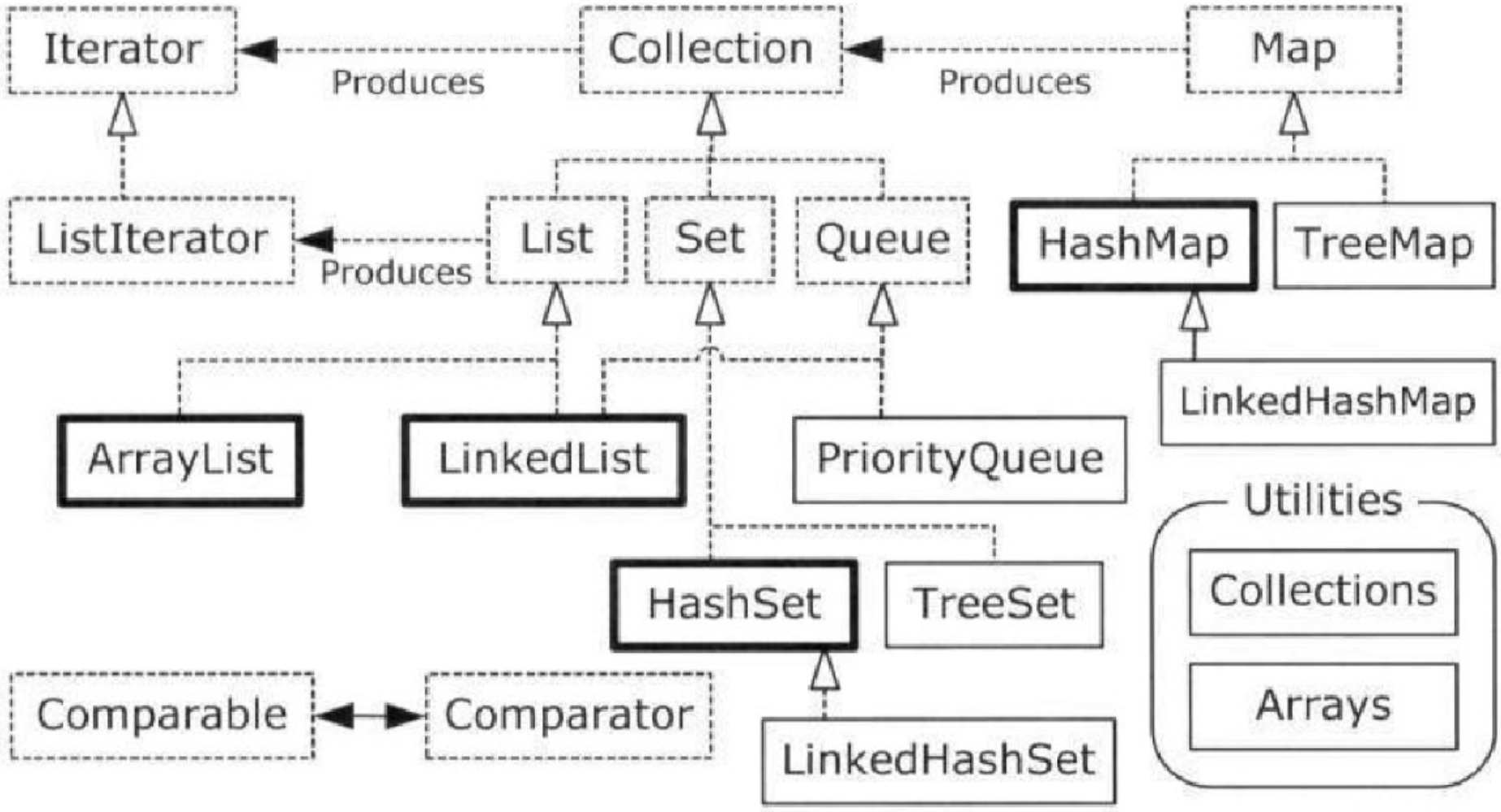
Importanza didattica

- Fornisce ottimi esempi di uso di composizione, ereditarietà, genericità
- Mette in pratica pattern di programmazione di interesse
- Impatta su alcuni aspetti del linguaggio da approfondire

JCF – struttura complessiva



JCF – struttura riorganizzata



JCF – alcuni aspetti generali

È complessivamente piuttosto articolato

- Un nostro obbiettivo è quello di isolare una sua sottoparte di interesse
- Identificando e motivando le funzionalità prodotte

Due tipi di collection, ognuna con varie incarnazioni

- `Collection` – contenitore di elementi atomici
  - 3 sottotipi: `List` (sequenze), `Set` (no duplicazioni), `Queue`
- `Map` – contenitore di coppie chiave-valore

Interfacce/classi di interesse:

- Interfacce: `Collection`, `List`, `Set`, `Iterator`, `Comparable`
- Classi collection: `ArrayList`, `LinkedList`, `HashSet`, `HashMap`
- Classi con funzionalità: `Collections`, `Arrays`

Una nota su eccezioni e JCF

Eccezioni: un argomento che tratteremo in dettaglio

Un meccanismo usato per gestire eventi ritenuti fuori dalla normale esecuzione (errori), ossia per dichiararli, lanciarli, intercettarli

JCF e eccezioni

- Ogni collection ha sue regole di funzionamento, e non ammette certe operazioni che richiedono controlli a tempo di esecuzione (ad esempio, certe collezioni sono immutabili, e non si può tentare di scriverci)
- Molti metodi dichiarano che possono lanciare eccezioni – ma possiamo non preoccuparcene per ora

Outline

Foreach

Costrutto foreach

- Abbiamo visto che può essere usato per iterare su un array in modo più astratto (compatto, leggibile)
  - `for(int i: array){...}`
- Java fornisce anche un meccanismo per usare il foreach su qualunque collection, in particolare, su qualunque oggetto che implementa l'interfaccia `java.lang.Iterable<X>`

`Iterable` e `Iterator`

- L'interfaccia `Iterable` ha un metodo per generare e restituire un (nuovo) `Iterator`
- Un iteratore è un oggetto con metodi `next()`, `hasNext()` (e `remove()`)
- Dato l'oggetto `coll` che implementa `Iterable<T>` allora il foreach diventa:
  - `for(T element: coll){...}`

Interfacce per l'iterazione

```java
package java.lang;
import java.util.Iterator;
public interface Iterable<T> {
    /**
     * Returns an iterator over a set of elements of type T.
     *
     * @return an Iterator.
     */
    Iterator<T> iterator();
}
```

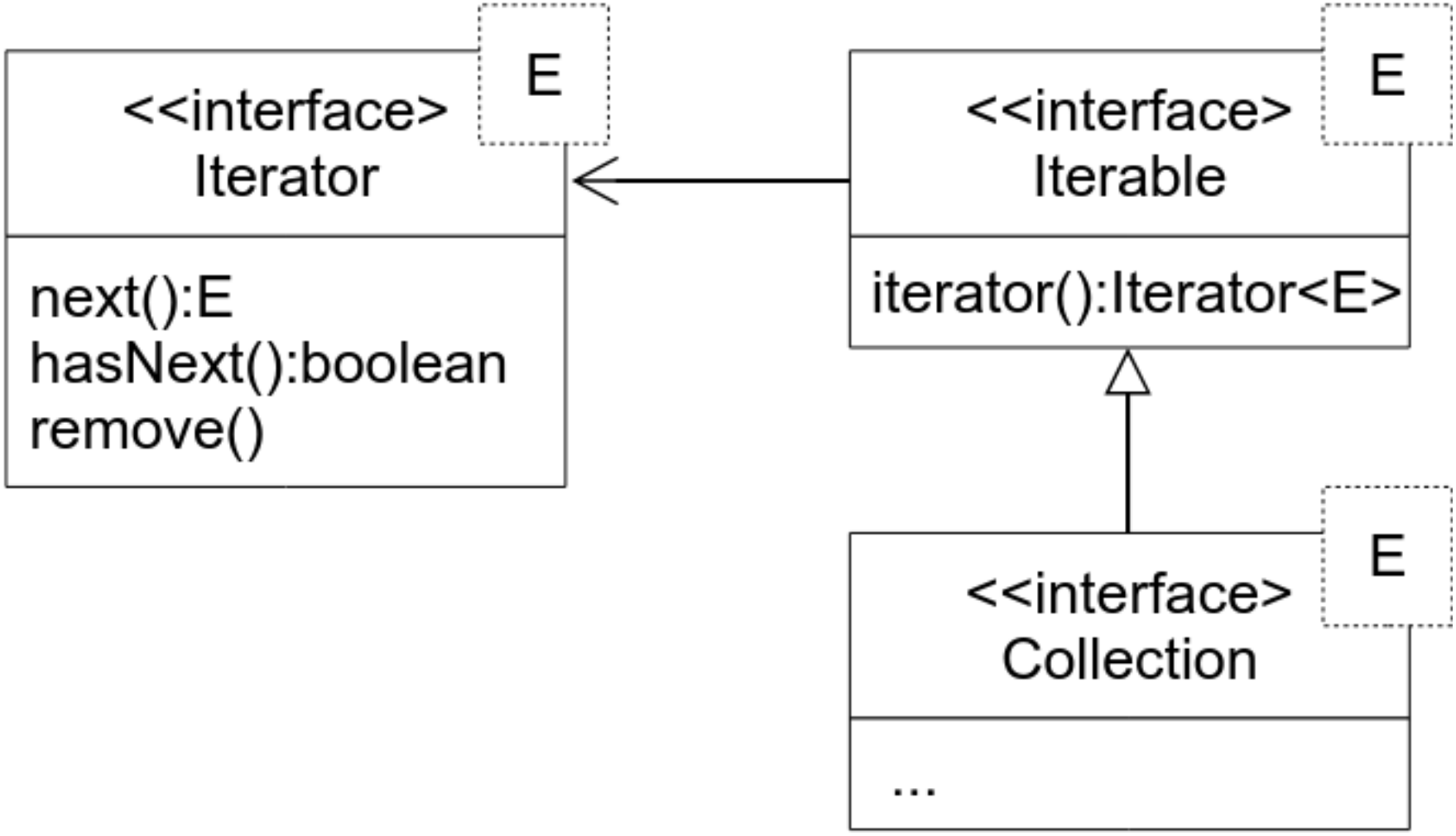
```java
package java.util;

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

```java
package java.util;

public interface Collection<E> implements Iterable<E> { .. }
```

Interfacce per l'iterazione – UML



Esempio di Iterable ad-hoc, e suo uso

```java
public class Range implements Iterable<Integer> {

  private final int start;
  private final int stop;

  public Range(final int start, final int stop) {
    this.start = start;
    this.stop = stop;
  }

  public java.util.Iterator<Integer> iterator() {
    return new RangeIterator(this.start, this.stop);
  }
}
```

```java
public class UseRange {
  public static void main(String[] s) {
    for (final int i : new Range(5, 12)) {
      System.out.println(i);
      // 5 6 7 8 9 10 11 12
    }
  }
}
```

Realizzazione del corrispondente iteratore

```
1  class RangeIterator implements java.util.Iterator<Integer> {
2
3    private int current;
4    private final int stop;
5
6    public RangeIterator(final int start, final int stop) {
7      this.current = start;
8      this.stop = stop;
9    }
10
11   public Integer next() {
12     return this.current++;
13   }
14
15   public boolean hasNext() {
16     return this.current <= this.stop;
17   }
18
19   public void remove() {
20   }
21 }
```

Iteratori e collezioni: preview

```java
import java.util.*;

public class UseCollectionIterator {

  public static void main(String[] s) {
    // Uso la LinkedList
    final LinkedList<Double> list = new LinkedList<>();
    // Inserisco 50 elementi
    for (int i = 0; i < 50; i++) {
      list.add(Math.random());
    }
    // Stampo con un foreach
    int ct = 0;
    for (double d : list) {
      System.out.println(ct++ + "\t" + d);
    }
    // 0 0.1023051360273742 3
    // 1 0.4318582138894327
    // 2 0.5239222319032795
    // ..
  }
}
```

Outline

Interfaccia `Collection`

Ruolo di questo tipo di dato

- È la radice della gerarchia delle collezioni
- Rappresenta gruppi di oggetti (duplicati/non, ordinati/non)
- Implementata via sottointerfacce (`List` e `Set`)

Assunzioni

- Definisce operazioni base valide per tutte le collezioni
- Assume implicitamente che ogni collezione abbia due costruttori
  - Senza argomenti, che genera una collezione vuota
  - Che accetta un `Collection`, dal quale prende tutti gli elementi
- Le operazioni di modifica sono tutte "opzionali"
  - potrebbero lanciare un `UnsupportedOperationException`
- Tutte le operazioni di ricerca lavorano sulla base del metodo `Object.equals()` da chiamare sugli elementi
  - questo metodo accetta un `Object`, influendo su alcuni metodi di `Collection`

Collection

```
1  public interface Collection<E> extends Iterable<E> {
2
3      // Query Operations
4      int size();                   // number of elements
5      boolean isEmpty();            // is the size zero?
6      boolean contains(Object o); // does it contain an element equal to o?
7      Iterator<E> iterator();       // yields an iterator
8      Object[] toArray();           // convert to array of objects
9      <T> T[] toArray(T[] a);       // puts in ‘a‘, or create new if too small
10
11     // Modification Operations
12     boolean add(E e);             // adds e
13     boolean remove(Object o);     // remove one element that is equal to o
14
15     // Bulk Operations
16     boolean containsAll(Collection<?> c);        // contain all elements in c
        ?
17     boolean addAll(Collection<? extends E> c);  // add all elements in c
18     boolean removeAll(Collection<?> c);          // remove all elements in c
19     boolean retainAll(Collection<?> c);          // keep only elements in c
20     void clear();                                // remove all element
21
22     // ...and other methods introduced in Java 8
23 }
```

Usare le collezioni

```java
public class UseCollection {
  public static void main(String[] s) {
    // Uso una incarnazione, ma poi lavoro sull'interfaccia
    final Collection<Integer> coll = new ArrayList<>();
    coll.addAll(Arrays.asList(1, 3, 5, 7, 9, 11)); // var-args
    System.out.println(coll); // [1, 3, 5, 7, 9, 11]

    coll.add(13);
    coll.add(15);
    coll.add(15);
    coll.remove(7);
    System.out.println(coll); // [1, 3, 5, 9, 11, 13, 15, 15]

    coll.removeAll(Arrays.asList(11, 13, 15));
    coll.retainAll(Arrays.asList(1, 2, 3, 4, 5));
    System.out.println(coll); // [1, 3, 5]
    System.out.println(coll.contains(3)); // true
    System.out.println(Arrays.toString(coll.toArray()));

    Integer[] a = new Integer[2];
    a = coll.toArray(a);
    System.out.println(Arrays.deepToString(a));
  }
}
```

Creare collezioni (immutabili) – Java 9+

```java
public class UseFactories {
  public static void main(String[] s) {
    // Metodi statici di creazione per Set e List immutabili

    final Set<Integer> set = Set.of(1, 2, 3, 4, 5, 6);
    System.out.println(set);

    final List<String> list = List.of("a", "b", "c", "a");
    System.out.println(list);

    final Set<String> set2 = Set.copyOf(list);
    System.out.println(set2);

  }
}
```

Set e List

Set

- Rappresenta collezioni senza duplicati
  - nessuna coppia di elementi porta `Object.equals()` a dare `true`
  - non vi sono due elementi `null`
- Non aggiunge metodi rispetto a `Collection`
- I metodi di modifica devono rispettare la non duplicazione

List

- Rappresenta sequenze di elementi
- Ha metodi per accedere ad un elemento per posizione (0-based)
- Andrebbe scandito via iteratore/foreach, non con indici incrementali
- Fornisce un list-iterator che consente varie operazioni aggiuntive

La scelta fra queste due tipologie non dipende da motivi di performance, ma da quale modello di collezione serva!

Set e List

```java
public interface List<E> extends Collection<E> {
    // Additional Bulk Operations
    boolean addAll(int index, Collection<? extends E> c);

    // Positional Access Operations
    E get(int index);                   // get at position index
    E set(int index, E element);        // set into position index
    void add(int index, E element); // add, shifting others
    E remove(int index);                // remove at position index

    // Search Operations
    int indexOf(Object o);              // first equals to o
    int lastIndexOf(Object o);          // last equals to o

    // List Iterators
    ListIterator<E> listIterator();             // iterator from 0
    ListIterator<E> listIterator(int index);  // ..from index

    // View
    List<E> subList(int fromIndex, int toIndex);
}
```

```java
public interface Set<E> extends Collection<E>{}
```

ListIterator

```java
package java.util;

public interface ListIterator<E> extends Iterator<E> {
    // Query Operations

    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();

    // Modification Operations

    void remove();
    void set(E e);
    void add(E e);
}
```

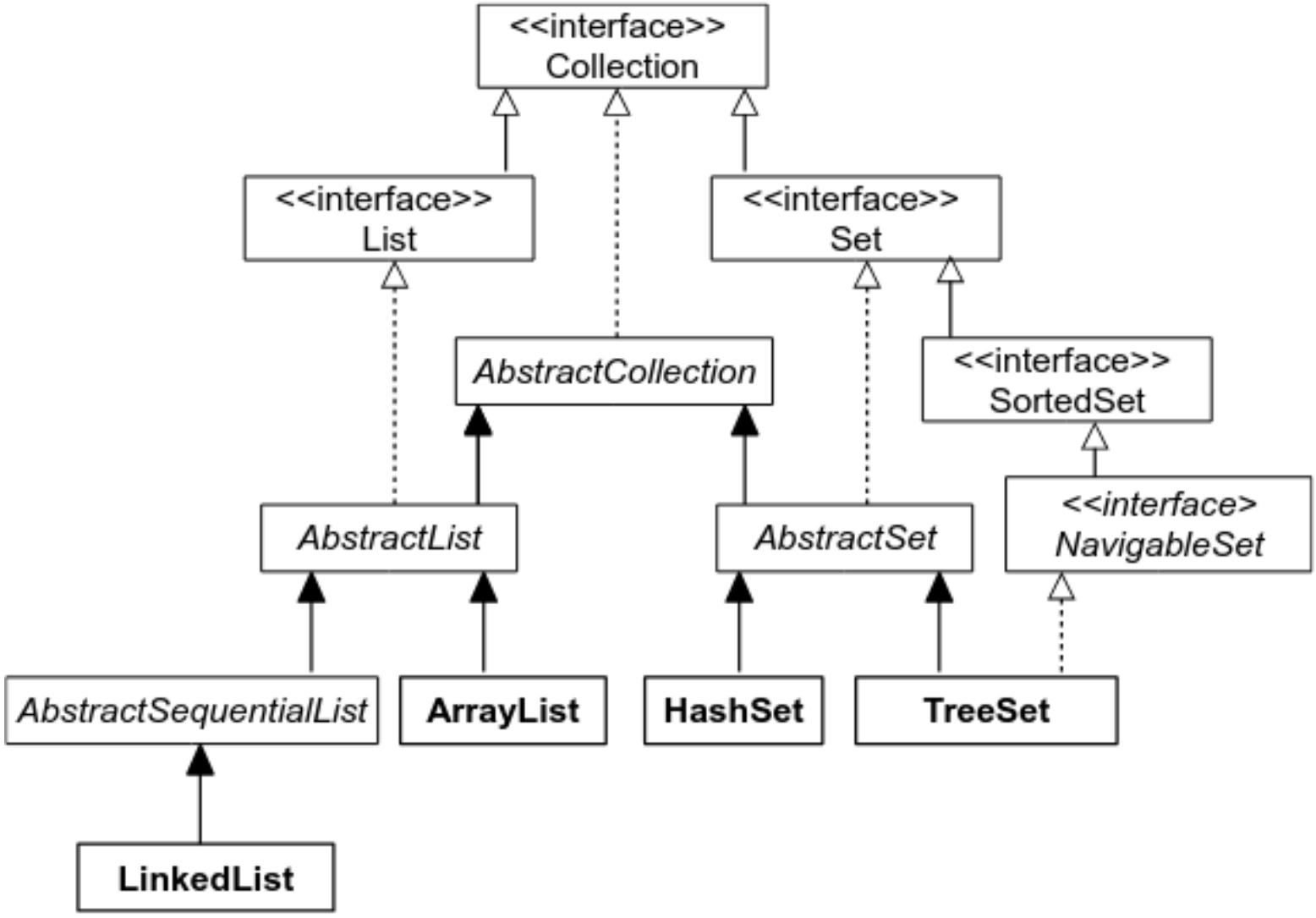
UseListIterator

```java

public class UseListIterator {
  public static void main(String[] s) {
    // Uso una incarnazione, ma poi lavoro sul List
    final List<Integer> list = new ArrayList<>();
    list.addAll(Arrays.asList(1, 3, 5, 7, 9, 11)); // var-args

    final ListIterator<Integer> it = list.listIterator();
    while (it.hasNext()) {
      it.add(it.next() + 1);
    }
    System.out.println(list); // [1, 2, 3, ..., 10, 11,12]
    while (it.hasPrevious()) {
      System.out.println("back: " + it.previous()); // 12 .. 1
    }
    for (final int i : list.subList(3, 10)) {
      System.out.println("forth - 3 to 10: " + i); // 4 .. 10
    }
  }
}
```

Implementazione collezioni – UML



Implementazione collezioni: linee guida generali

Una modalità di progettazione da ricordare

- Interfacce: riportano le funzionalità definitorie del concetto
- Classi astratte: fattorizzano codice comune alle varie implementazioni
- Classi concrete: realizzano le varie implementazioni

Nel codice cliente..

- In variabili, argomenti, tipi di ritorno, si usano le interfacce
- Le classi concrete solo nella `new`, a parte casi molto particolari
- Le classi astratte non si menzionano praticamente mai, solo eventualmente per chi volesse costruire una nuova implementazione

Implementazione collezioni – Design space

Classi astratte

- `AbstractCollection`, `AbstractList`, e `AbstractSet`
- Realizzano "scheletri" di classi per collezioni, corrispondenti alla relative interfacce
- Facilitano lo sviluppo di nuove classi aderenti alle interfacce

Un esempio: `AbstractSet`

- Per set immutabili, richiede solo di definire `size()` e `iterator()`
- Per set mutabili, richiede anche di ridefinire `add()`
- Per motivi di performance si potrebbero fare ulteriori override

Classi concrete.. fra le varie illustreremo:

- `HashSet`, `TreeSet`, `ArrayList`, `LinkedList`
- La scelta riguarda quasi esculsivamente esigenze di performance

Outline

Implementazioni di Set

Caratteristiche dei set

- Nessun elemento duplicato (nel senso di `Object.equals()`)

→ Il problema fondamentale è il metodo `contains()`, nelle soluzioni più naive (con iteratore) potrebbe applicare una ricerca sequenziale, e invece si richiedono in genere performance migliori

Approccio 1: `HashSet`

Si usa il metodo `Object.hashCode()` come funzione di hash, usata per posizionare gli elementi in uno store di elevate dimensioni

Approccio 2: `TreeSet`

Specializzazione di `SortedSet` e di `NavigableSet`. Gli elementi sono ordinati, e quindi organizzabili in un albero (red-black tree) per avere accesso in tempo logaritmico

HashSet

Idea di base: tecnica di hashing (via `Object.hashCode()`)

- Si crea un array di elementi più grande del necessario (p.e. almeno il 25% in più), di dimensione `size`
- Aggiunta di un elemento `e`
  - lo si inserisce in posizione `e.hashCode() % size`
  - se la posizione è già occupata, lo si inserisce nella prima disponibile
  - se l'array si riempie va espanso e si deve fare il rehashing
- Ricerca di un elemento `f`
  - si guarda a partire da `f.hashCode() % size`, usando `Object.equals()`
  - La funzione di hashing deve evitare il più possibile le collisioni
- Risultato: scritture/letture sono $O(1)$ ammortizzato

Dettagli interni

- Realizzata tramite `HashMap`, che approfondiremo in futuro

Costruttori di HashSet

```
1  public class HashSet<E>
2      extends AbstractSet<E>
3      implements Set<E>, Cloneable, java.io.Serializable {
4
5      // Set vuoto, usa hashmap con capacità 16
6      public HashSet() {...}
7
8      // Set con elementi di c, usa hashmap del 25% più grande di c
9      public HashSet(Collection<? extends E> c) {...}
10
11     // Set vuoto
12     public HashSet(int initialCapacity, float loadFactor) {...}
13
14     // Set vuoto, loadFactor = 0.75
15     public HashSet(int initialCapacity) {...}
16
17     /* Gli altri metodi di Collection seguono... */
18 }
```

equals() e hashCode()

La loro corretta implementazione è cruciale

- Le classi di libreria di Java sono già OK
- `Object` uguaglia lo stesso oggetto e l'hashing restituisce la posizione in memoria..
- .. quindi nuove classi devono ridefinire `equals()` e `hashCode()` opportunamente

Quale funzione di hashing?

- oggetti `equals` devono avere lo stesso `hashCode`
- non è detto il viceversa, ma è opportuno per avere buone performance di `HashSet`
- si veda ad esempio: `http://en.wikipedia.org/wiki/Java_hashCode()`
- Eclipse/VSCode forniscono la generazione di un `hashCode` (e `hashCode`) ragionevoli (ce ne sono di migliori: `djb2`, `murmur3`)

Esempio: Person pt.1

```java
public class Person {

  final private String name;
  final private int year;
  final private boolean married;

  public Person(String name, int year, boolean married) {
    this.name = name;
    this.year = year;
    this.married = married;
  }

  public boolean isMarried() {
    return this.married;
  }

  public String getName() {
    return this.name;
  }

  public int getYear() {
    return this.year;
  }
```

Esempio: Person pt.2

```
1   public String toString() {
2     return this.name + ":" + this.year + ": marr-" + this.married;
3   }
4
5   public int hashCode() {
6     return Objects.hash(name, year);
7   }
8
9   public boolean equals(Object obj) {
10    if (this == obj) {
11      return true;
12    }
13    if (!(obj instanceof Person)) {
14      return false;
15    }
16    Person other = (Person) obj;
17    return Objects.equals(name, other.name) && year == other.year;
18  }
19 }
```

UseHashSetPerson

```java
public class UseHashSetPerson {
  public static void main(String[] s) {
    // HashSet è un dettaglio, lavorare sempre su Set!
    final Set<Person> set = new HashSet<>();

    // Agggiungo 4 elementi
    set.add(new Person("Rossi", 1960, false));
    set.add(new Person("Bianchi", 1980, true));
    set.add(new Person("Verdi", 1972, false));
    set.add(new Person("Neri", 1968, false));
    System.out.println(set);

    // Testo presenza/assenza di 2 elementi
    final Person p1 = new Person("Rossi", 1960, false);
    final Person p2 = new Person("Rossi", 1961, false);
    System.out.println("Check "+p1+" found "+set.contains(p1));
    System.out.println("Check "+p2+" found "+set.contains(p2));

    // Iterazione: nota, fuori ordine rispetto all'inserimento
    for (final Person p : set) {
      System.out.println("Iterating: "+p+" hash = "+p.hashCode());
    }
  }
}
```

TreeSet<E>

Specializzazione NavigableSet (e SortedSet)

- Assume che esista un ordine fra gli elementi
- Quindi ogni elemento ha una sua posizione nell'elenco
- Questo consente l'approccio dicotomico alla ricerca
- Consente funzioni addizionali, come le iterazioni in un intervallo

Due realizzazione: con criterio di ordinamento interno o esterno

1. O con elementi che implementano direttamente `Comparable`
   - Nota che, p.e., `Integer` implementa `Comparable<Integer>`
2. O attraverso un `Comparator` esterno fornito alla `new`

Implementazione TreeSet

- Basata su red-black tree (albero binario bilanciato)
- Tempo logaritmico per inserimento, cancellazione, e ricerca

Comparazione “interna” agli elementi

```java
public interface Comparable<T> {
    /* returns: 0 (this == o), positive (this > o)
                          negative (this < o) */
    public int compareTo(T o);
}
```

```java
class Integer extends Number implements Comparable<Integer>{ ... }
class String extends Object implements Comparable<String>,..{ ... }
// >100 classi delle librerie di Java seguono questo approccio
```

```java
public class Person implements Comparable<Person>{
    ...
    // Esempio di implementazione di compareTo:
    // - ordine di anno di nascita, e poi per nome..
    public int compareTo(Person p){
      return (this.year != p.year)
          ? this.year - p.year
          : this.name.compareTo(p.name);
    }
}
```

Esempi di comparazione interna

```
1  import java.util.*;
2
3  public class UseComparison {
4
5    public static void main(String[] s) {
6
7      System.out.println("abc vs def: " + "abc".compareTo("def")); // neg
8      System.out.println("1 vs 2: " + Integer.valueOf(1).compareTo(2));// neg
9
10     final Person p1 = new Person("Rossi", 1960, false);
11     final Person p2 = new Person("Rossi", 1972, false);
12     final Person p3 = new Person("Bianchi", 1972, false);
13     final Person p4 = new Person("Bianchi", 1972, true);
14
15     System.out.println(p1 + " vs " + p2 + ": " + p1.compareTo(p2)); // pos
16     System.out.println(p2 + " vs " + p3 + ": " + p2.compareTo(p3)); // pos
17     System.out.println(p3 + " vs " + p4 + ": " + p3.compareTo(p4)); // zero
18
19     System.out.println(new TreeSet<Integer>(Arrays.asList(4,3,2,1)));
20      // 1,2,3,4
21   }
22 }
```

Interfacce SortedSet e NavigableSet

```java
public interface SortedSet<E> extends Set<E> {
    Comparator<? super E> comparator();
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);   // fino a toElement
    SortedSet<E> tailSet(E fromElement); // da toElement
    E first();
    E last();
}
```

```java
public interface NavigableSet<E> extends SortedSet<E> {
    E lower(E e);   // Elemento prima di e
    E floor(E e);   // Elemento prima di e (e incluso)
    E ceiling(E e); // Elemento dopo e (e incluso)
    E higher(E e);  // Elemento dopo e
    E pollFirst();  // Torna ed elimina il primo se esiste
    E pollLast();   // Torna ed elimina l'ultimo se esiste
    NavigableSet<E> descendingSet();  // Set con ordine invertito
    Iterator<E> descendingIterator();   // .. e relativo iteratore
    NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                           E toElement,   boolean toInclusive);
    NavigableSet<E> headSet(E toElement, boolean inclusive);
    NavigableSet<E> tailSet(E fromElement, boolean inclusive);
}
```

UseTreeSetPerson: comparazione interna

```java
public class UseTreeSetPerson {
  public static void main(String[] s) {

    final List<Integer> l = Arrays.asList(new Integer[] { 10, 20, 30, 40 });
    // TreeSet è un dettaglio, lavorare sempre sull'interfaccia

    final NavigableSet<Person> set = new TreeSet<>();
    set.add(new Person("Rossi", 1960, false));
    set.add(new Person("Bianchi", 1980, true));
    set.add(new Person("Verdi", 1972, false));
    set.add(new Person("Neri", 1972, false));
    set.add(new Person("Neri", 1968, false));

    // Iterazione in ordine, poi al contrario, poi fino al 1970
    for (final Person p : set) {
      System.out.println("Iterating: " + p + " hash = " + p.hashCode());
    }
    for (final Person p : set.descendingSet()) {
      System.out.println("Opposite iteration: " + p);
    }
    final Person limit = new Person("", 1970, false);
    for (final Person p : set.headSet(limit, false)) {
      System.out.println("Iterating to 1970: " + p);
    }
  }
}
```

Costruttori di TreeSet, e comparatore "esterno"

```
1  public class TreeSet<E> extends AbstractSet<E>
2      implements NavigableSet<E>, Cloneable, java.io.Serializable{
3
4      // Set vuoto di elementi confrontabili
5      public TreeSet() {...}
6
7      // Set vuoto con comparatore fornito
8      public TreeSet(Comparator<? super E> comparator) {...}
9
10     // Set con gli elementi di c, confrontabili tra loro
11     public TreeSet(Collection<? extends E> c) {...}
12
13     // Set con gli elementi di c, e che usa il loro ordering
14     public TreeSet(SortedSet<E> s) {...}
15
16     /* Seguono i metodi di NavigableSet e SortedSet */
17 }
```

```
1  public interface Comparator<T> {
2      // 0 if o1 == o2, neg if o1 < o2, pos is o1 > o2
3      int compare(T o1, T o2);
4  }
```

Definizione di un comparatore esterno

```java
import java.util.Comparator;

/* Implementa la politica di confronto esternamente a Persona */

public class PersonComparator implements Comparator<Person> {

  // Confronto prima sul nome, poi sull'anno
  public int compare(Person o1, Person o2) {
    return o1.getName().equals(o2.getName())
        ? o1.getYear() - o2.getYear() // Integer.compare(o1.getYear(), o2.getYear())
        : o1.getName().compareTo(o2.getName());
  }
}
```

UseTreeSetPerson2: comparazione esterna

```java
import java.util.*;

public class UseTreeSetPerson2{

  public static void main(String[] s){

      // TreeSet è un dettaglio, lavorare sempre sull'interfaccia
      final Set<Person> set =
          new TreeSet<>(new PersonComparator());

      set.add(new Person("Rossi",1960,false));
      set.add(new Person("Bianchi",1980,true));
      set.add(new Person("Verdi",1972,false));
      set.add(new Person("Neri",1972,false));
      set.add(new Person("Neri",1968,false));

      // Iterazione in ordine
      for (final Person p: set){
          System.out.println(p);
      }
    }
}
```

Perché il tipo `Comparator<? super E>`

Data una classe `SortedSet<E>` il suo comparatore ha tipo `Comparator<? super E>`, perché non semplicemente `Comparator<E>`?

È corretto

- `Comparator` ha metodi che hanno `E` solo come argomento
- quindi l'uso di `Comparator<? super E>` è una generalizzazione di `Comparator<E>`

È utile

- Supponiamo di aver costruito un comparatore per `SimpleLamp`, e che questo sia usabile anche per tutte le specializzazioni successivamente costruite (è la situazione tipica)
- Anche un `SortedSet<UnlimitedLamp>` deve poter usare il `Comparator<SimpleLamp>`, ma questo è possibile solo grazie al suo tipo atteso `Comparator<? super E>`

Un esempio di design con le collezioni

Implementare questa interfaccia che modella un archivio persone

```java
import java.util.Set;

public interface Archive {

  void add(String nome, int annoNascita, boolean sposato);

  void remove(String nome, int annoNascita);

  int size();

  Set<String> allMarried();
}
```

Una soluzione con HashSet

```
1  public class ArchiveImpl implements Archive {
2
3    private final Set<Person> set = new HashSet<>();
4
5    public void add(String nome, int annoNascita, boolean sposato) {
6      this.set.add(new Person(nome,annoNascita,sposato));
7    }
8
9    public void remove(String nome, int annoNascita) {
10     this.set.remove(new Person(nome,annoNascita,false));
11   }
12
13   public int size() {
14     return this.set.size();
15   }
16
17   public Set<String> allMarried() {
18     final Set<String> newset = new HashSet<>();
19     for (final Person p: this.set) {
20       if (p.isMarried()) {
21         newset.add(p.getName());
22       }
23     }
24     return newset;
25   }
26 }
```

Scenario d'uso dell'archivio

```java
public class UseArchive {

  public static void main(String[] args) {
    final Archive arc = new ArchiveImpl();
    arc.add("Rossi", 1960, false);
    arc.add("Bianchi", 1980, true);
    arc.add("Verdi", 1972, true);
    arc.add("Neri", 1968, false);
    arc.remove("Neri", 1968);

    System.out.println(arc.size()); // 3
    System.out.println(arc.allMarried()); // [Bianchi,Verdi]
  }

}
```