Corso di Laurea in Ingegneria Informatica e Automatica

Programmazione orientata agli oggetti

Paolo Liberatore, Massimo Mecella

Java Collections Framework

Sommario

- Introduzione a Java Collections Framework
- Tipi generici
- Interfacce del JCF
- Classi del JCF
- Collezioni ordinate

Java Collections Framework

Il **Java Collections Framework (JCF)** è una libreria formata da un insieme di interfacce e di classi che le implementano per lavorare con collezioni di oggetti.

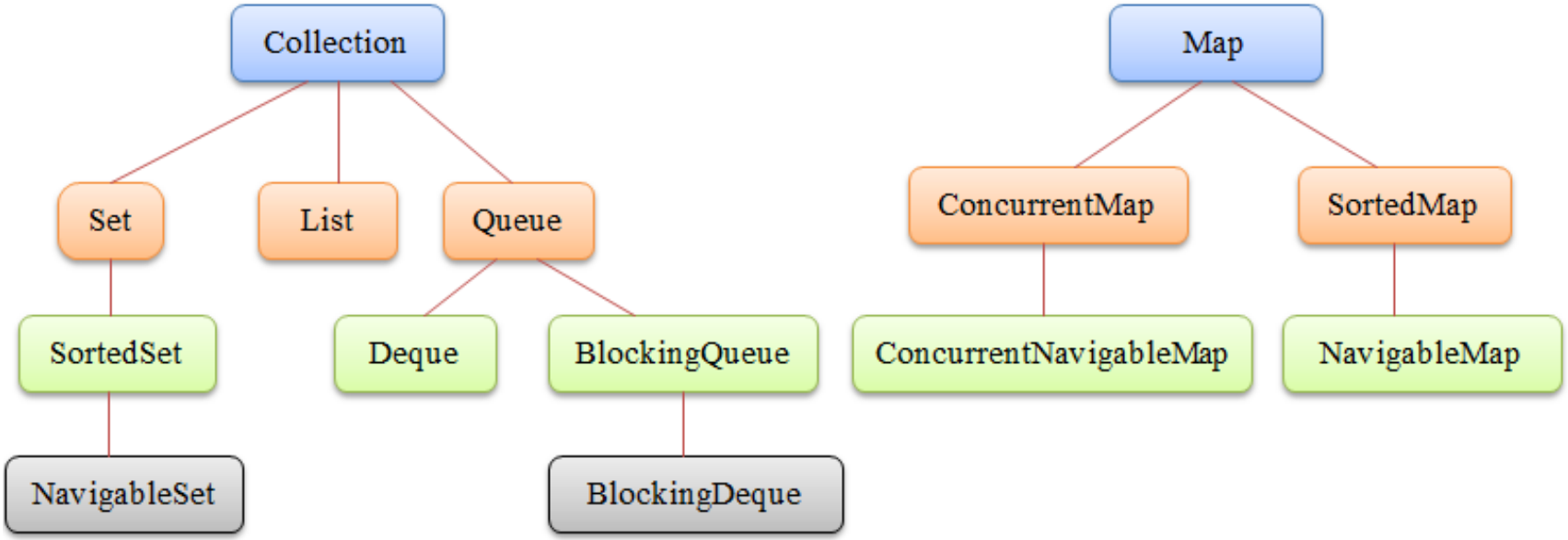
- Interfacce: rappresentano vari tipi di collezioni di uso comune
- Implementazioni: classi concrete che implementano le interfacce di cui sopra, utilizzando strutture dati efficienti
- Algoritmi: funzioni che realizzano algoritmi di uso comune, quali algoritmi di ricerca e di ordinamento su oggetti che implementano le interfacce del JCF

Java Collections Framework

Vantaggi nell'uso del JCF

- Generalità: permette di modificare l'implementazione di una collezione senza modificare i clienti
- Interoperabilità: permette di utilizzare (e farsi utilizzare da) codice realizzato indipendentemente dal nostro.
- Efficienza: le classi che realizzano le collezioni sono ottimizzate per avere buone prestazioni

Interfacce del JCF

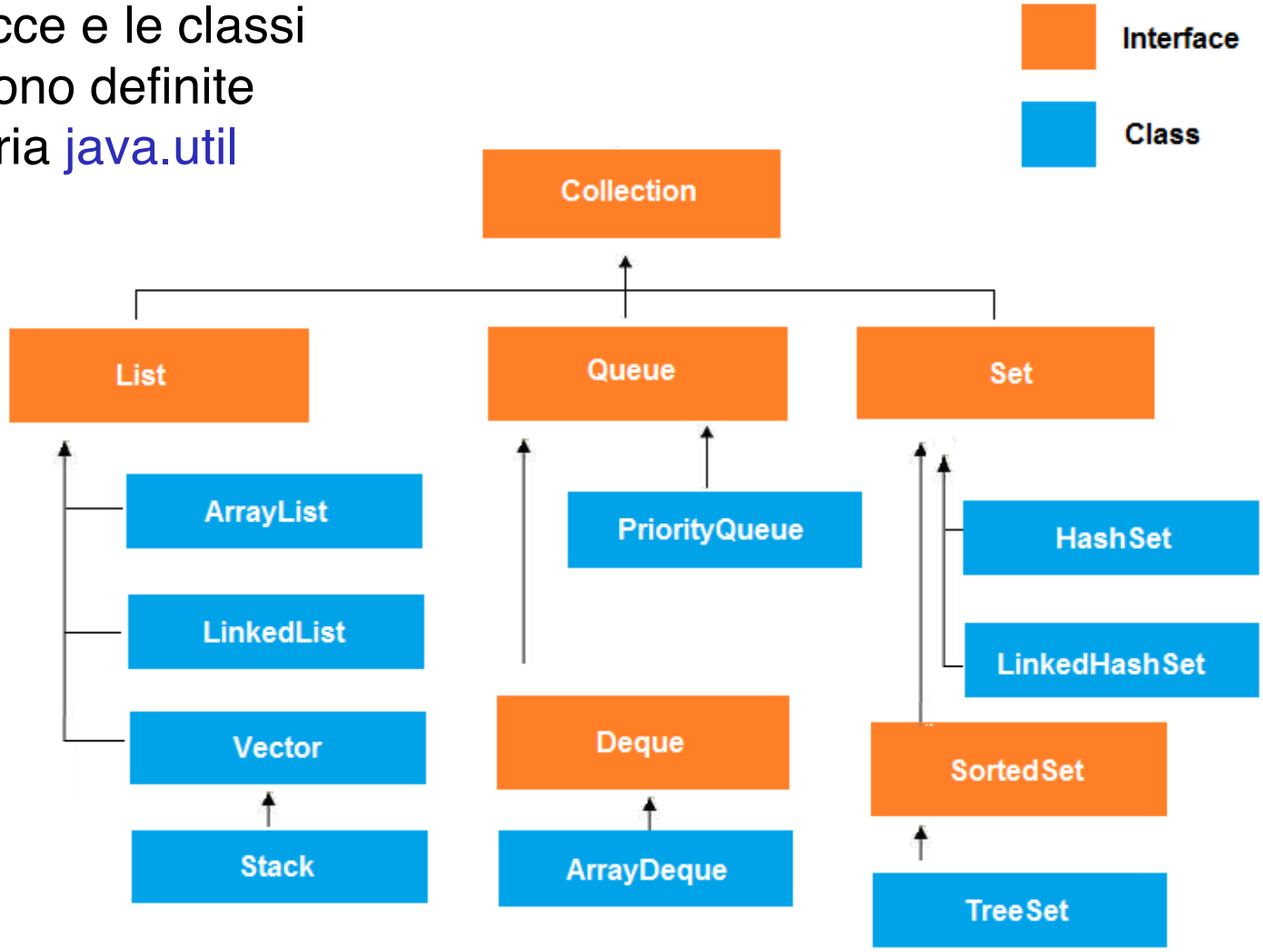


Interfacce del JCF

- Set: insiemi
  - SortedSet: insiemi ordinati
- List: liste (o sequenze)
- Queue: code
  - Deque: coda a doppia uscita
- Map: mappe <chiave, valore>
  - SortedMap: mappe con chiavi ordinate
- …

Interfacce e classi del JCF

Le interfacce e le classi del JCF sono definite nella libreria java.util



Tipi Generici

I **tipi generici** sono usati in Java per definire classi, interfacce e metodi parametriche rispetto ad un tipo di dati su cui operano.

I tipi generici (introdotti dalla versione 5) hanno sostituito la consuetudine di usare la classe Object come contenitore di qualsiasi oggetto.

Definizione di Tipi Generici

La definizione di un tipo generico C rispetto ad un altro tipo T viene indicata usando la notazione <T> nella dichiarazione di C

Esempio

```
public class C<T> {
    ....
}
```

Definizione di Tipi Generici

Un tipo generico C<T> può usare al suo interno il tipo T per riferire variabili e argomenti dei metodi.

Esempio

```
class C<T> {
    private T info;
    public C(T info) { this.info=info; }
    public T getInfo() { return info; }
    public void setInfo(T info) { this.info=info; }
}
```

Uso di Tipi Generici

Un tipo generico C<T> deve essere usato istanziando il tipo di riferimento T con una classe specifica.

Esempio

```
C<String> c1 = new C<String>("ciao");
C<Persona> c2 = new C<Persona>(new Persona(…));

String s = c1.getInfo();
Persona p = c2.getInfo();
….
```

Uso di Tipi Generici

I tipi generici vengono usati nel JCF per definire collezioni di oggetti di un certo tipo T.

Il vantaggio rispetto ad usare collezioni di oggetti della classe Object è che tramite i tipi generici possiamo effettuare un controllo di tipo a tempo di compilazione.

Ad esempio, in una collezione di persone (cioè di oggetti di tipo Persona) possiamo avere solo oggetti compatibili con la classe Persona (cioè appartenente a Persona o ad una delle sue sottoclassi).

Interfaccia Collection

```
public interface Collection<E> {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); // Optional
    boolean remove(Object element); // Optional
    Iterator<E> iterator();

    boolean equals(Object o);
    ....
```

Interfaccia Collection

```
public interface Collection<E> {
    ....

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // Optional
    boolean removeAll(Collection<?> c); // Optional
    boolean retainAll(Collection<?> c); // Optional
    void clear(); // Optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Interfaccia Set

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

}
```

L'interfaccia Set non definisce alcuna nuova funzione rispetto a Collection.

Interfaccia List

```
public interface List<E> extends Collection<E> {
    boolean add(int index, E element); // Optional
    E get(int index);
    E set(int index, E element); // Optional
    int indexOf(Object o);
    int lastIndexOf(Object o);
    boolean remove(int index);

    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index)

    boolean addAll(int index, Collection<? extends E> c); // Optional
    List<E> subList(int fromIndex, int toIndex);
}
```

Interfaccia Iterator

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

public interface ListIterator<E> extends Iterator<E> {
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void set(E e); // Optional
}
```

Iteratori

Un **iteratore** è un oggetto di una classe che implementa l'interfaccia Iterator che rappresenta un cursore con il quale scandire una collezione alla quale esso è associato.

public Iterator iterator() in Collection restituisce un iteratore con il quale scandire la collezione oggetto di invocazione

Un iteratore è sempre associato ad un oggetto collezione e la sua realizzazione dipende dalla classe concreta che implementa la collezione.

Il programmatore può usare gli iteratori (tramite l'interfaccia Iterator) senza conoscere i dettagli implementativi delle classi che implementano le collezioni e gli iteratori stessi.

Uso degli iteratori

Esempio

```
Collection<E> c = ... // collezione di oggetti di tipo E
...
Iterator<E> it = c.iterator(); // iteratore per la collezione c
while (it.hasNext()) { // finche' il cursore non e' all'ultimo elemento
    E e = it.next(); // poni l'elemento corrente in e ed avanza
    ...    // processa l'elemento corrente (denotato da e)
}
```

Uso degli iteratori

Esempio

```
List<E> c = ... // lista di oggetti di tipo E
...
ListIterator<E> it = c.listIterator(c.size()); // iteratore per la lista c
                // inizializzato per puntare all'ultimo elemento della lista
while (it.hasPrevious()) { //finche' il cursore non e' al primo elemento
    E e = it.previous(); // poni l'elemento precedente in e
                                    // e sposta il cursore indietro
    ...    // processa l'elemento corrente (denotato da e)
}
```

Mappe

Una **mappa** è una struttura dati che serve a memorizzare coppie **(chiave,valore)**.

- La **chiave** serve ad accedere alla coppia ed è univoca all'interno di una mappa.
- Il **valore** serve a memorizzare informazioni associate alla chiave.

Esempi di mappe:

- dizionario: coppie (parola, significato)
- rubrica: coppie (nominativo, indirizzo e telefono)
- funzione discreta: coppie (x,y)

Interfaccia Map

```
public interface Map<K,V> {
    int size();
    boolean isEmpty();
    void clear(); // Optional
    boolean equals(Object o);

    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value); // Optional
    V remove(Object key); // Optional

    Set<K> keySet();
    Collection<V> values();
}
```

Collezioni ordinate

Il Java Collections Framework prevede anche la possibilità di definire collezioni ordinate.

- SortedSet: per rappresentare insiemi di oggetti ordinati (non sono ammessi oggetti ripetuti).
- SortedMap: per rappresentare mappe ordinate per chiave.

L'ordinamento nelle collezioni consente una maggiore efficienza degli algoritmi.

Il criterio di ordinamento nelle collezioni ordinate è stabilito dalle interfacce Comparable o Comparator.

Collezioni ordinate: Comparable

I tipi della collezione implementano l'interfaccia Comparable

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

compareTo() confronta l'oggetto di invocazione this con l'oggetto passato come parametro o, restituendo un intero negativo (-1) se this è più piccolo di o, 0 se sono uguali, o un intero positivo (1) se this è più grande di o.

Esempio:

public class Persona implements Comparable …

Collezioni ordinate: Comparator

Si definisce una classe che implementa Comparator.

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

compare() confronta gli oggetti o1 e o2, restituendo un intero negativo (-1) se o1 è più piccolo di o2, 0 se sono uguali, o un intero positivo (1) se o1 è più grande di o2.

Uso di collezioni ordinate

Le classi che implementano SortedSet o SortedMap sono tipicamente dotate di due costruttori:

- Costruttore senza argomenti: che basa l'ordinamento degli oggetti nella collezione sul fatto che questi implementino l'interfaccia Comparable
- Costruttore con un argomento di tipo Comparator: che basa l'ordinamento degli oggetti nella collezione sull'ordinamento indotto dalla funzione compare del comparatore passato al costruttore

Uso di collezioni ordinate

Esempio

```
class Persona implements Comparable<Persona> {
    ….
    int compareTo(Persona o) { …. }  // ordinamento per cognome e nome

}

class PersonaComparatorEta implements Comparator<Persona> {

    int compare(Persona p1, Persona p2) { …. } // ordinamento per eta'

}
```

Uso di collezioni ordinate

Esempio

```
SortedSet<Persona> s1 = new TreeSet<Persona>();
SortedSet<Persona> s2 = new TreeSet<Persona>(
                                        new PersonaComparatorEta() );

// s1: insieme di persone ordinate per cognome e nome
// s2: insieme di persone ordinate per età
```

Implementazione delle collezioni

Strutture dati usate nelle implementazioni delle interfacce del JCF

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Implementazione delle collezioni

- Collection: nessuna implementazione specifica.

- Set: classe HashSet basata su **tavola hash**,
- SortedSet: classe TreeSet basata su **albero di ricerca bilanciato**

- List: classe ArrayList basata su **array dinamico**
- List: classe LinkedList basata su **lista collegata doppia**

- Map: classe HashMap basata su **tavola hash**
- SortedMap: classe TreeMap basata su **albero di ricerca bilanciato**

Esercizio

Modificare l'implementazione dell'esercizio Biblioteca (L3) usando una lista (interfaccia List del JCF) per rappresentare gli autori di un libro.

Modificare la classe Biblioteca usando degli insiemi (interfaccia Set del JCF) per implementare gli elenchi degli autori e dei libri.