Java Collection Framework

Java Collection Framework

- Prima di Java 2 erano disponibili poche classi container, soprattutto se riferite alla libreria STL del C++

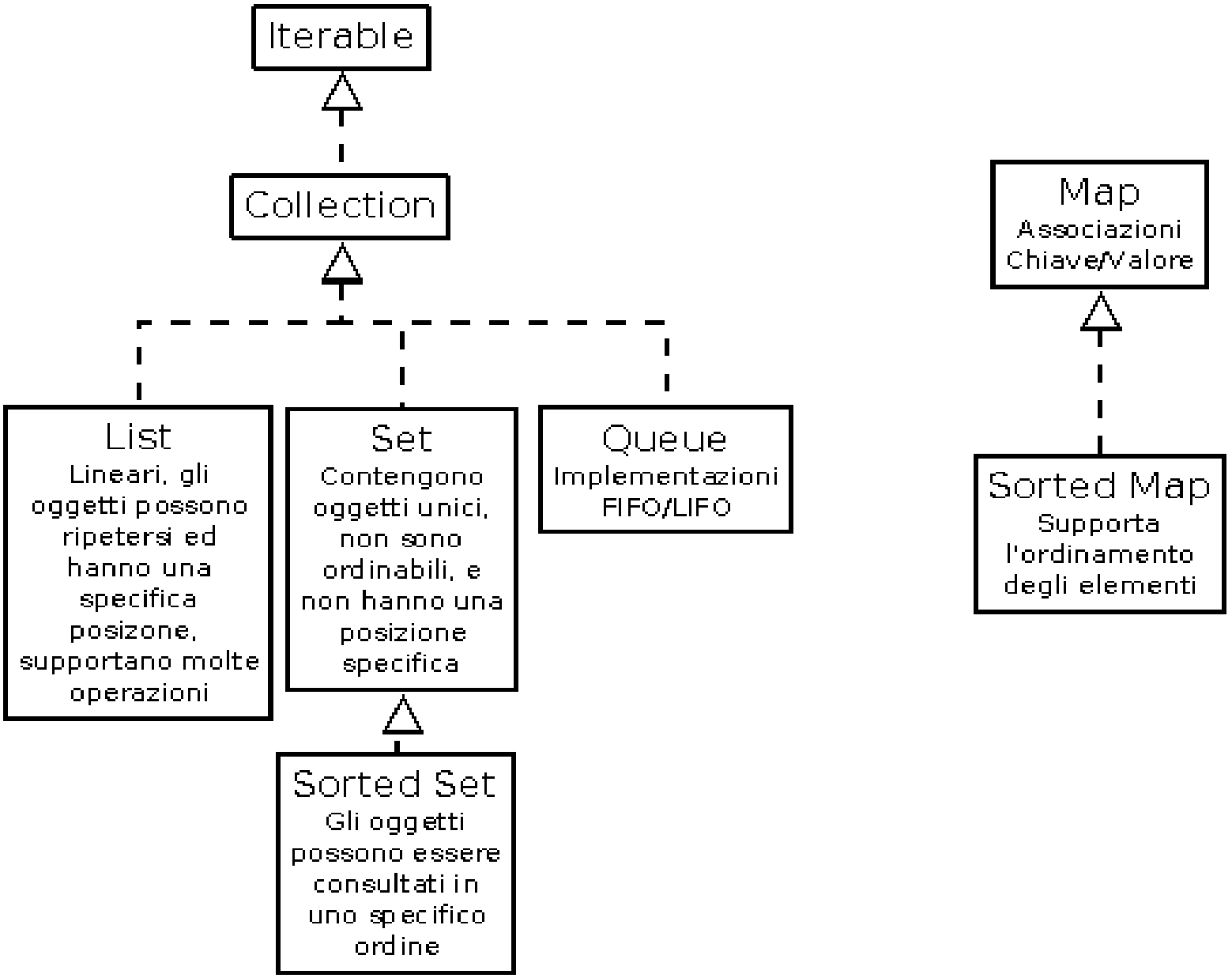
`Vector, Stack, Hashtable, BitSet, Enumeration`

La libreria standard Java si è dotata via via di diverse Collection, ovvero classi idonee ad ospitare collezioni di oggetti diversamente strutturate, caratterizzate dalle loro interfacce, arrivando a costituire la Java Collection Framework (package java.lang.util)

Un framework è convenzionalmente un insieme di classi che realizzano funzionalità avanzate e meccanismi di composizione/funzionamento, che possono essere facilmente adoperati da sottoclassi, senza dovere riscrivere i meccanismi base (es Swing)

- Gli elementi della JCF si suddividano in quelli che hanno come antenata la interfaccia **Collection**, e quelli che implementano la interfaccia **Set**

Le interfacce della JCF



La interfaccia Collection

- La interfaccia fondamentale per i tipi Collection è si chiama Collection (parametrica), che ha due metodi fondamentali, “add” e “iterator”

```
public interface Collection<E>
{
   boolean add(E element);
   Iterator<E> iterator();
  //== altri metodi . . .
}
```

Add vale true se l'elemento aggiunto “cambia” la collection (nel caso di set un oggetto già presente con cambia la collection

A partire da Java 5 L'interfaccia Iterator è parametrica:

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

Invocando il metodo next() si visitano tutti gli elementi della collezione, se non ci sono più elelemtni da visitare il metodo `next` solleva l'eccezione `NoSuchElementException.` `hasNext` restituisce true se l'oggetto iterator ha ancora elementi da visitare.

A partire da Java 5, c'è' un maniera alternativa e più compatta per realizzare il loop di un iterartore

```
Collection<E> c = . . .;

Iterator<E> iter = c.iterator();

while (iter.hasNext())

{

String element = iter.next();

//== CODICE

}
```

```
for (E element : c)
{
 //== CODICE
}
```

- L'ordine con cui vengono restituiti gli elementi dall'iteratore dipendono dal tipo di Collection. Se le collection prevedono una forma di ordinamento, gli elementi vengono restituiti in ordine crescente (es ArrayList, Array partendo dall'indice 0)
- Nel caso degli Set, es HashSet, l'ordine è casuale ma ovviamente viene garantito che tutti gli elementi vengono restituiti dall'iteratore

- IL metodo remove rimuove dalla collezione l'ultimo elemento restituito da next. Non è possibile rimuovere due elementi consecutivi invocando due volte “remove”

```
it.remove();
it.next();
it.remove(); // Ok
```

- Nella interfaccia Collection ci sono dei metodi di utilità generale per ciascuna delle collection concrete

`int size()` *numero di elementi della coll.*

`boolean isEmpty()` *true se la coll. è vuota*

`boolean contains(Object obj)`

true se la coll. contiene obj (confronto fatto con equals)

`boolean containsAll(Collection<?> c)`

true se la collection contiene tutti gli elementi di c

`boolean add(Object other)`

aggiunge un object other alla coll. this.

`boolean addAll(Collection<? extends E> From)`

aggiunge tutti gli oggetti di una altra coll (rest true se la collection cambia)

`boolean remove(Object obj)`
rimuove o dalla collection (rest. True se l'oggetto era presente e quindi rimosso)
`boolean removeAll(Collection<?> c)`
rimuove dalla collection this tutti gli oggetti presenti in c (rest. True se la coll. cambia ovvero se almeno un oggetto viene rimosso)
`void clear()` *rimuove dalla collection tutti gli oggetti*
`boolean retainAll(Collection<?> c)`
rimuove dalla collection this tutti gli oggetti TRANNE quelli che sono presenti in c. rest true se la collection cambia

`Object[] toArray()`
conversione in array di object

- I metodi add e remove hanno argomento di tipo Object mentre non sono parametriche.
- Se non riescono ad aggiungere o a togliere l'oggetto passato come argomento (anche perché di tipo inatteso) restituiscono false
- In alcuni casi può essere utile passare tipi diversi a questi metodi

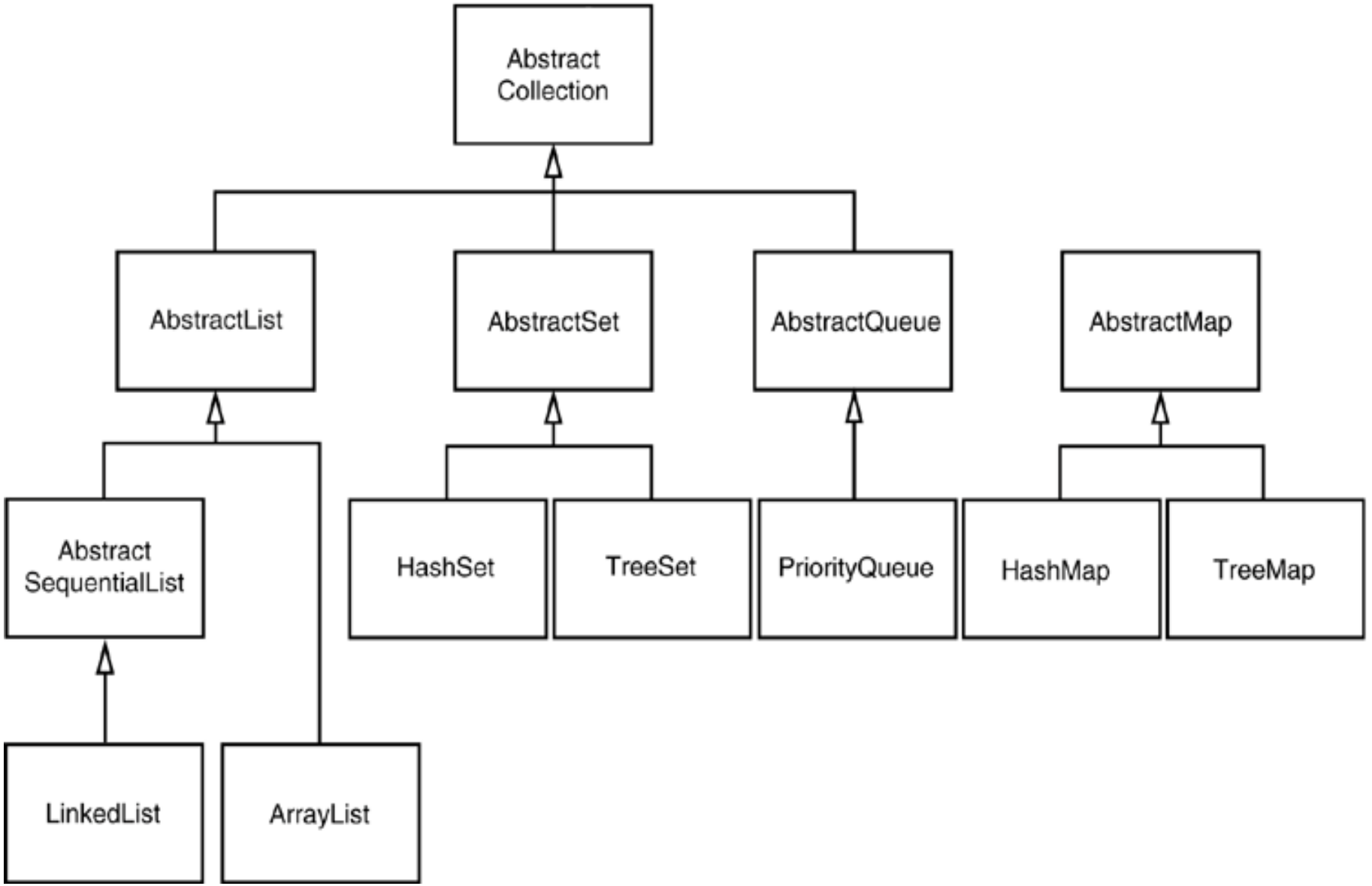
Implementazioni

- Per facilitare l'implementazione delle collection concrete è stata introdotta una classe astratta intermedia, e molti metodi dell'interfaccia sono implementati tramite i metodi della classe astratta “AbstractCollection” (tranne iterator e size())

```
public abstract class AbstractCollection<E>
                                    implements Collection<E>
{     . . .
   public abstract Iterator<E> iterator();
    public boolean contains(Object obj)
   {
      for (E element : c) // calls iterator()
         if (element.equals(obj))

            return = true;
      return false;
   }
  ...
}
```

Le classi della JCF



AbstractCollection

AbstractList

AbstractSequentialList

AbstractSet

AbstractQueue

AbstractMap

- Le classi astratte forniscono i metodi fondamentali per le implementazioni delle classi concrete

LinkedList

ArrayList

HashSet

TreeSet

PriorityQueue

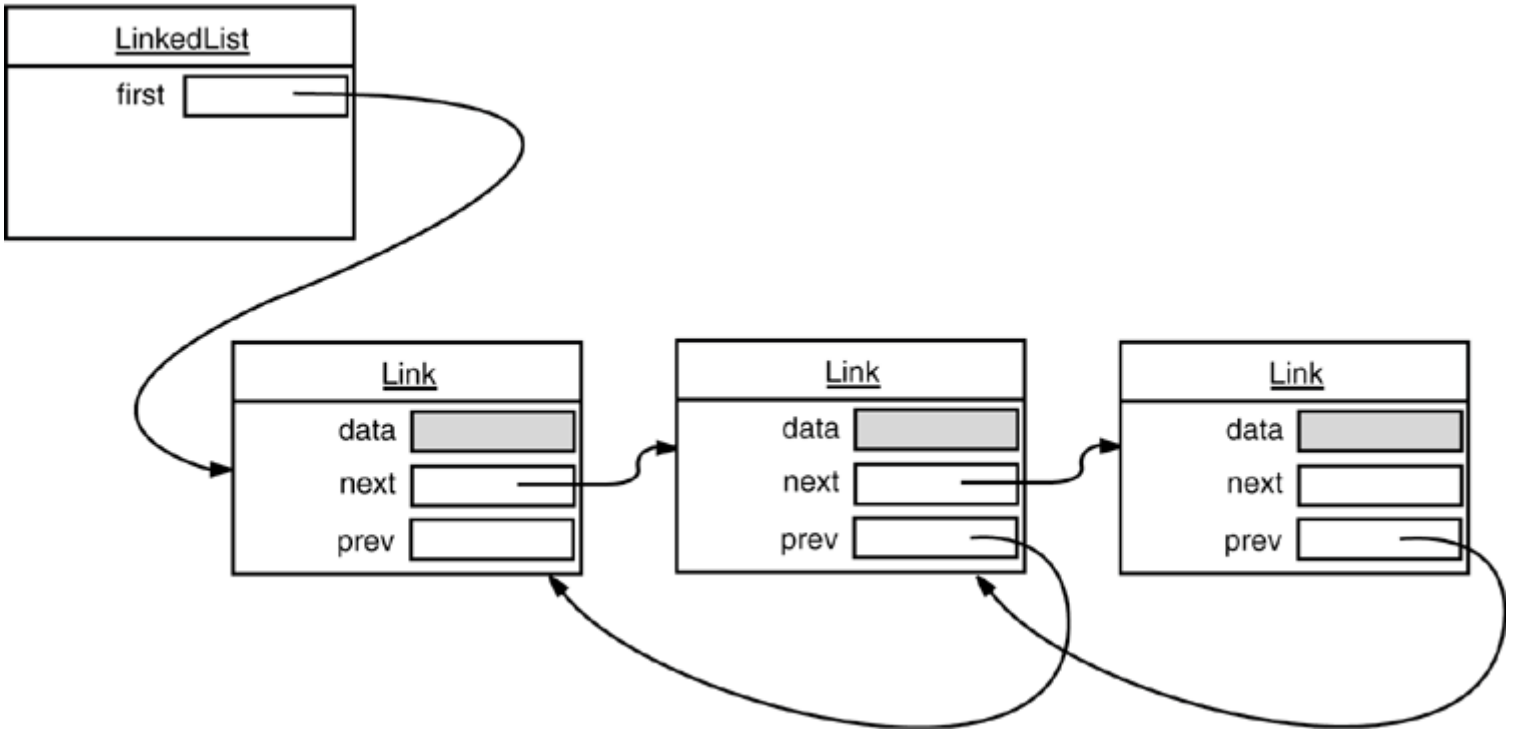
HashMap

TreeMap

- La JFC mette a disposizione un gruppo di classi concrete per le implementazioni di collezioni più comuni

LinkedLIst

- Liste doppiamente linkate



Il metodo **`LinkedList.add`** aggiunge l'elemento alla fine della lista.

Per aggiungere un elemento in una posizione qualsiasi, la JFC fornisce una sotto interfaccia List Iterator che contiene un metodo add, che consente l'inserimento nel corso di una iterazione.

```
ListIterator<E> extends Iterator<E>

{
   void add(E element); //== NB non rest.boolean
 //== elemento precedente all'attuale (navigazone
 //== bidirezionale)
   E previous();
   boolean hasPrevious();
   set(E element) //== rimpiazza l'elemento
    //== corrente con element
  listIterator(int n) //== iterazione dalla posizione n

} (ES)
```

- Sebbene con l'iteratore listIterator si riesce ad operare sulle posizioni degli elementi, se le posizioni degli oggetti sono importanti, si dovrebbe adoperare la collection ArrayList, piuttosto che List per motivi di efficienza.
- Pur disponendo dei metodi “get(n)”, per accedere ad un elemento in posizione n la list inizia una iterazione sempre dalla posizione iniziale

List

`ListIterator<E> listIterator()`

`ListIterator<E> listIterator(int index)`

Restituisce un listIterator a partire dalla posizione n (dopo n invocazioni di next)

`void add(int i, E element)` aggiunge un elemento alla posizione i

`void addAll(int i, Collection<? extends E> elements)` aggiunge tutti gli elementi di una collezione a partire dalla posizione i

`E remove(int i)` Rimuove e restituisce un elemento che si trova nella posizione i

`E set(int i, E element)` Restituisce l'elemento che si trova nella posizione i, rimpiazzandolo con l'elemento passato come argomento.

`int indexOf``(Object element)` Restituisce la posizione di un dato elemento oppure -1 se l'elemento non c'è' nella collezione.

`int lastIndexOf``(Object element)` Restituisce l'ultima posizione di un elemento in una collezione, oppure -1 se l'elemento non viene trovato

List Iterator

`void add(E newElement)` aggiunge un elemento prima della posizione corrente

`void set(E newElement)` restituisce l'ultimo elemento visitato con `next` o `previous` con new element. Solleva "`IllegalStateException`" se la lista è stata modificata prima della chiamata a `next` o `previous`.

`boolean hasPrevious()` .

`E previous()` L'oggetto precedente. Si solleva l'ecc, `NoSuchElementException` se ci si trova all'inizio della lista.

`int nextIndex()` L'indice dell'elemento che verrebbe restituito da un chiamata del metodo next()

`int previousIndex()` L'indice dell'elemento che verrebbe restituito da una chiama a previous

Linked List

`LinkedList()` Costruisce una linklist vuota

`LinkedList(Collection<? extends E> elements)` Costruisce una linked list e la riempie degli elementi della collection passata come argomento.

`addFirst(E element)`

`addLast(E element)`

Aggiungono un elemento in testa ed in coda alla lista.

`E getFirst()`

`E getLast()`

Restituiscono il primo e l'ultimo elemento della Lista

`E removeFirst()`

`E removeLast()`

Rimozione del primo e dell'ultimo elemento

ArrayList

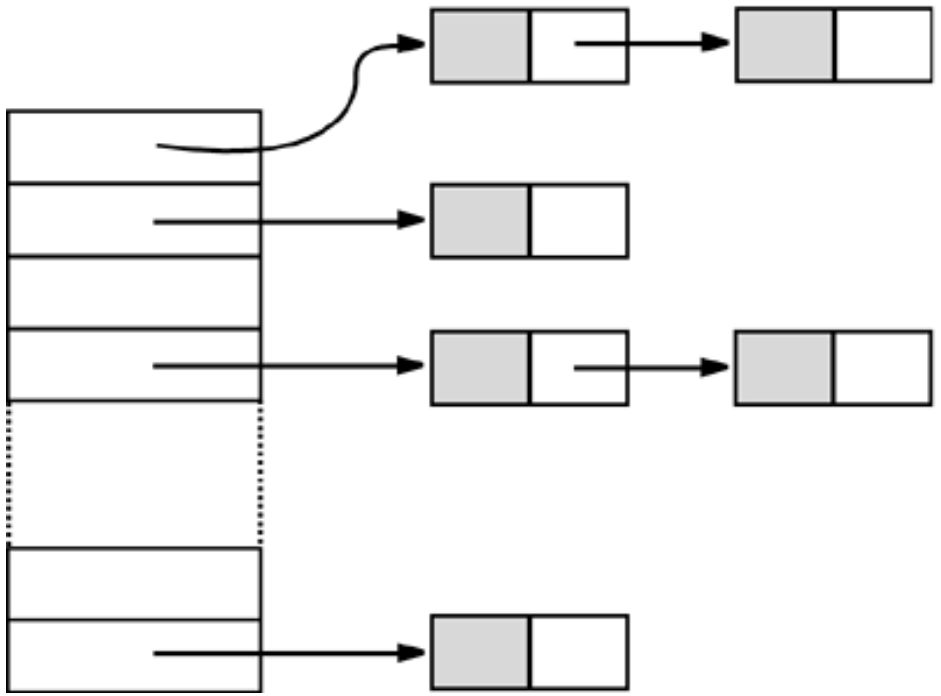
- Gli ArrayList incapsulano un array dinamico di oggetti, si occupano della allocazione e della riallocazione dinamica in maniera trasparente per l'utilizzatore, quando vengono aggiunti oggetti, cancellati, ecc.
- Sono simili ai Vector, con la differenza che non sono thread safe (non sono sincronizzati), e vanno usato quando non è richiesta la sincronizzazione

- Gli ArrayList sono molto più efficienti per la ricerca *posizionale* degli oggetti rispetto le Liste.
- Implementano l'interfaccia RandomAccess, che richiede la realizzazione di un algoritmo efficiente per la ricerca degli elementi in base alla loro posizione

HashSet

- Nelle collection tipo Set gli oggetti non hanno una posizione (non sono ordinati), ma sono organizzati in strutture dati che consentano la loro reperibilità in maniera più efficiente
- Gli HashSet sono implementazioni di insiemi, che conservano gli oggetti in tabelle tipo HashTable. Una HashTable associa ad ogni oggetto un numero intero univoco, e si presenta come un array di liste linkate, ciascuna delle quali rappresenta un bucket.

HASH TABLE



- Bucket

  Calcolo della posizione

  Bucket dell'oggetto=(HashCode dell'oggetto % numero di bucket). L'oggetto va poi confrontato con gli altri elementi della lista (per vedere se già presente), se il bucket non è vuoto

- Gli hashcode degli oggetti sono calcolati col metodo HashCode (override del metodo di obkect),. Se due oggetti sono “equals”, allora per un corretto inserimento in una hashSet (no duplicazioni), *dovrebbero* avere lo stesso hashCode
- Con gli HashSet il numero di bucket è possibile fissarlo (dovrebbe oscillare tra il 75% ed il 150% elementi da conservare)
- E' possibile specificare il leadFactor, ovvero il rapporto tra il numero di elementi ed il numero di bucket: quando questo viene superato, l'hash table viene ricreata e si aumenta il numero di bucket per soddisfare il load factory (ES)

Hash Set

`HashSet()` Costruttore di un hash Set vuoto

`HashSet(Collection<? extends E> elements)`
costruisce un hash set e ripone in essa gli oggetti di una altra Collection

`HashSet(int initialCapacity)`

`HashSet(int initialCapacity, float loadFactor)`

costruisce un hash set specificando una capacità e un loadfactory iniziali

Tree Set

- I TreeSet estendono gli HashSet, includendo anche un *ordinamento* degli elementi. Indipendentemente da come vengono inseriti, l'iterarotre del tree set restituisce gli oggetti in maniera ordinata.
- Il tree set ha la forma di un albero rosso/nero
- Aggiungere un elmento è una operazione meno efficiente rispetto agli HashSet, ma comunque più efficiente rispetto una linkedlist che preservi l'ordinamento. Per un tree set che contiene n elementi, sono richieste log(n) confronti, rispetto ai agli n confronti di una lista ordinata

- Affinché gli oggetti inseriti in una tree Set siano ordinati, devono essere confrontabili, ovvero devono implementare l'interfaccia Comparable

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

Il metodo compareTo deve restituire 0 se gli oggetti sono uguali, 1 se l'oggetto è se segue nella relazione di ordinamento di quello cui deve confrontarsi (è maggiore), -1 se è minore (ES)

- E' possibile implementare il comparatore esternamente come estensione dell'interfaccia “Comparator”

```
public interface Comparator<T>
{
   int compare(T a, T b);
}
```

Compare deve restituire 0 se gli oggetti `a` e `b` sono uguali, un numero `>0` se `a>b`, un numero `<0` se `a<b`

- Il comparatore è possibile costruirlo al volo come istanza di una classe interna anonima (ES)

```
SortedSet<Employee> ss = new TreeSet<Employee>
   (new Comparator<Employee>() {
       public int compare(Employee a, Employee b)
       {
          int name_cfr=a.name.compareTo(b.name);
          if (name_cfr==0) return
           ((int)a.salary-(int)b.salary);
          else return name_cfr;
       }
  });
```

- La funzione HashCode, per il suo utilizzo di indicizzazione delle tabelle hash, dovrebbe garantire che i codici hash siano calcolati in maniera opportuna, garantendo una distribuzione uniforme dei valori hash per gli oggetti istanziati
- Se si adopera la tecnica della scomposizione in fattori primi, si può rischiare di andare in over flow, per questo si può combinare gli hashCode dei membri della classe con l'operatore XOR bit a bit (^)

- Gli hash code si calcolano in base al valore dei campi in un determinato istante. Se questo cambia dopo l'inserimento di un elemento in una HashSet, può accadere che l'oggetto non venga piu' classificato come appartenente all'insieme (! contains) ES

TreeSet

- **`treeSet()`**

  Costruttore di un TreeSet vuoto.

- **`treeSet(Collection<? extends E> elements)`**

  Costruisce un tree set e gli aggiunge tutti gli elementi di una collezione “elements”.

Priority Queues

•Una “priority queue” restituisce gli elementi ordinati, anche quando sono stati inseriti in qualsiasi ordine. Questo vale per l'iteratore, ma anche per i metodi add e remove : viene eliminato sempre l'elemento in cima alla coda.

•Gli elementi sono inseriti in una struttura dati tipo Heap, che è un albero binario dove le operazioni add e remove fanno in modo che l'elemento piu' piccolo vada ad essere messo nella posizione root, senza bisogno di dover ordinare tutti gli elementi (Heap Sorting).

Come per il tree set, anche le PriorityQueues ospitano oggetti che implementano l'interfaccia Comparable, oppure bisogna passargli il metodo Comparator come argomento.

Un tipico utilizzo delle priority queues è quello degli schedulatori, per la gestione automatica delle priorità dei job inseriti e prelevati dalla coda.

PriorityQueue

`PriorityQueue(int initialCapacity)`

costruttore di una PriorityQueue con una data capacità iniziale.

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

costruttore di una PriorityQueue con una data capacità iniziale e con un dato comparatore

Map

- Le Map sono particolari collezioni caratterizzate dal fatto che ad ogni oggetto conservato è associato un valore. A differenza degli HashSet e degli HashCode, la chiave viene stabilita dall'utilizzatore e può essere di qualsiasi tipo.
- Le Map non sono in realtà classificate come Collection, ma implementano una interfaccia distinta
- Permettono di accedere rapidamente ad uno oggetto tramite la sua chiave

- Esistono due tipi di Map concrete:

  HashMap e TreeMap

- La HashMap converte in codice intero le chiavi (“hashing”), la TreeMap utilizza un ordinamento delle chiavi per localizzare ed inserire gli elementi
- Le HashMap sono più efficienti come set (verifica ed inserimento degli elementi), mentre le TreeMap offrono la caratteristica dell'ordinamento

Utilizzo.

```
Map<String, Employee>staff=
     new HashMap<String,Employee>();
Employee harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
```

Generalmente le chiavi sono stringhe (ma possono essere qualsiasi oggetto). Per accedere all'oggetto bisogna usare la stessa chiave adoperata nell'immagazzinamento

```
String s = "987-98-9996";
Employee e = staff.get(s);
```

La stessa Map può restituire l'insieme delle chiavi come collection di tipo set

```
Set<String> keys = map.keySet();
for (String key : keys)
{
//==
}
```

Se in corrispondenza di una chiave non è conservato alcun valore nella mappa, get restituisce null.

Le chiavi sono uniche, il metodo set ammette che la stessa chiave venga utilizzata per due elementi in questo caso il secondo rimpiazza il primo nella mappa.

Il metodo “remove” rimuove un elemento data una chiave.

Sebbene la JFC non consideri Le Map come implementazioni della interfaccia Collection, si possono ottenere i valori di una Map come collection

```
Set<K> keySet()
//== L'insieme delle chiavi

Collection<K> values()
//== collection dei valori

Set<Map.Entry<K, V>> entrySet()
//== L'insieme delle coppie chiave valore, sono
gli oggetti della classe Interna Map.Entry
statica, e può essere usato come qualsiasi
oggetto che estende l'interfaccia Collection
```

Per consultare contemporaneamente chiave e valori di una Map si può utilizzare il metodo entrySet

```
for (Map.Entry<String, Employee> entry :
staff.entrySet())
{
String key = entry.getKey();
Employee value = entry.getValue();
//…
}
```

Versioni specializzate di Map

WeakHashMap

Cosa accade se in una HashMap si annulla la referenza alla chiave? Nulla, perché nella hashMap le chiavi sono riferimento di tipo “strong”.

Le strong reference (riferimenti ordinari degli oggetti), impediscono la cancellazione della referenza da parte del Garbage Collector fino a che l'oggetto è raggiungibile da una strong reference.

Classificazione di Reachability degli oggetti (raggiungibilità) degli oggetti in un proramma Java

1. Strong Reachability. La raggiungibilità ordinaria, gli oggetti sono creati ed in uso e non vengono cancellati da GC fino a che c'e' una (strong) reference che punta all'oggetto.

2. Soft Reachability. Un oggetto è una soft Reachable quando non è raggiungibile da nessuna strong reference (o weak), ma solo da soft reference. il GC lo reclamerà quando riterrà necessario liberare la memoria (sicuramente prima di sollevare una eccezione OutOfMemory). Le Soft reference possono essere create tramite la classe SoftReference.

3. Weak Reachability. Un oggetto è weak reachable se è raggiungibile solo da weak reference. Il GC lo reclama appena possibile.

4. Phantom Reachability E' lo stato della referenza dopo la applicazione del metodo Finalize

5. Unreachable Un oggetto non raggiungibile

Un oggetto è quindi cancellabile dal GC se è raggiungibile da weak reference.

La WeakHashMap è una versione di HaspMap che coopera con il GarbageCollector. Se vengono eliminate tutte le referenze ad un oggetto *esternamente* alla WeakHashMap, viene eliminata anche la refererenza presente nella WeakHashMap. Le referenze delle chiavi nella HashMap sono gestite tramite oggetti di tipo “Weakreference”

Le String non sono adatte ad essere chiavi un una WeakhaspMap, perché vengono gestite in aree di memorie particolari dalla JVM, (la string reference pool), dove gli oggetti stringa sono in pratica sempre strong reference (immutabili): quando si crea una stringa si controlla se una uguale esiste, e viene restituito lo stesso riferimento

Linked HashMap

- Le linked hash map rircordano l'ordine con cui vengono inserite le coppie chiavi valore, e l'iteratore restiuisce gli elementi in questo ordine
- Quando si usa get o put su questo tipo di hashmap, l'elemento prescelto viene posto automaticamente all'ultimo posto, in questo modo alla prossima chiamata dell'iteratore verrà scorso per ultimo

Identity Hash Maps (JDK 1.4)

Nelle `IdentityHashMap` i valori hash delle chiavi non cono calcolate tramite il metodo `hashCode` degli oggetti chiave, ma tramite il metodo `System.identityHashCode` che è usato nella classe Object.hashCode per calcolare il valore hash a partire dall'indirizzo di memoria in assenza di overriding del metodo. Per il controllo di identità non si usa equal ma “==”

In questo modo oggetti differenti, anche se con uguale contenuto, vengono considerati diversi. E' utile quando si devono distinguere gli oggetti effettivamente istanziati, come nel caso degli algoritmi di serializzazione

View e wapper

- Esistono dei metodi delle classi della JFC in grado di offrire versioni diverse di collection, chiamate “view”, per esempio il metodo keySet delle Map, che restituisce l'inseme delle chiavi di una mappa.
- Le view si ottengono attraverso le classi astratte che fanno parte delle JFC, ed implementano le interfacce attraverso metodi esistenti (non costruiscono ex novo la collection).

- Esistono delle classi speciali in grado di fornire una versione “collection” di alcune strutture dati, chiamate wrapper

```
Card[] cardDeck = new Card[52];
List<Card> cardList = Arrays.asList(cardDeck);
```

Il metodo restituisce un tipo List che permette di manipolare l'array contenuto secondo i metodi dell'interfaccia List, ma non di *estenderlo* (no ArrayList)->unsupported execption

La classe Collections

- La classe Collections contiene diversi metodi di utilità per le classi della JCF

  es. Collections.nCopies(n, anObject)

Restituisce un oggetto che implementa l'interfaccia List che crea n copie di anObject

List<String> settings = Collections.nCopies(100, "DEFAULT");

In realà l'oggetto è conservato una sola volta (è una view)

Collections.singleton(anObject)

Restituisce una view di un oggetto che implementa l'interfaccia Set come immutabile “single element” Il fatto che si tratta di una view non produce costi di

ricostruzione. Esistono gli analoghi `singletonList` e `singletonMap`

Subrange

E' possibile selezionare delle porzioni di Collection come sub-range (sub list, sub array, ecc)

List group2 = staff.subList(10, 20);

Il primo indice è incluso, il secondo no

Tutte le operazioni che si effettuano sulle sub-list hanno effetto sull'intera Lista

group2.clear();

In questo modo tutti gli elementi presenti in group2 “scompaiono “ dalla lista staff

Per le mappe e gli insiemi ordinati, si usa l'ordinamento e non la posizione degli elementi per formare i sub range.

Metodi della interfaccia SortedSet:

```
subSet(from, to)
headSet(to)
tailSet(from)
```

Metodi equivalenti delle Map

```
subMap(from, to)
headMap(to)
tailMap(from)
```

restituiscono viste delle entry delle mappe in cui le rispettive chiavi ricadono nei subrange.

ALGORITMI e JCF

- Un grande vantaggio della JFC è quella di offrire una serie di algoritmi applicabile a diverse classi dell JFC ed alle loro estensioni
- Esempio, un algoritmo per trovare il massimo di una qualsiasi classe eche estede Collections di tipo comparable

```
public static <T extends Comparable> T max(Collection<T> c)
{
   if (c.isEmpty()) throw new NoSuchElementException();
   Iterator<T> iter = c.iterator();
   T largest = iter.next();
   while (iter.hasNext())
   {
      T next = iter.next();
      if (largest.compareTo(next) < 0)
         largest = next;
   }
   return largest;
}
```

```
List<String> staff = new LinkedList<String>();
// fill collection . . .;
Collections.sort(staff);
```

Il metodo presume che l'oggetto cui è applicato implementi l'interfaccia Comparable. E' possibile specificare un comparatore ad hoc per specifiche esigenze di ordinamento

```
Comparator<Item> itemComparator = new
   Comparator<Item>()
   {
      public int compare(Item a, Item b)
      {
         return a.partNumber - b.partNumber;
      }
   });
Collections.sort(items, itemComparator);
```

I

E

E'possibile specificare un ordinamento in ordine inverso, eventualmente specificando un comparatore particolare

```
Collections.sort(staff, Collections.reverseOrder())
Collections.sort(items,
          Collections.reverseOrder(itemComparator))
```

La classe Collections class has an algorithm "shuffle" che permuta in maniera random gli elementi di una lista, basandosi su un generatore di numeri casuali

```
ArrayList<Card> cards = . . .;
Collections.shuffle(cards)
```

- La classe Collections offre anche un metodo di ricerca basata sulla ricerca binaria, che presuppone che gli elementi sia ordinati. Il metodo Binarysearch

  presuppone che la collezione sia ordinata, è possibile pero' passare un comparatore particolare che deve essere lo stesso usato per l'ordinamento

```
i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, comparator);
```

```
static <T extends Comparable<? super T>>
      T min(Collection<T> elements)

static <T extends Comparable<? super T>>
      T max(Collection<T> elements)

static <T> min(Collection<T> elements,Comparator<? super T> c)

static <T> max(Collection<T> elements,Comparator<? super T> c)

static <T> void copy(List<? super T> to, List<T> from)
```

copia tutti gli elementi da una lista ad una altra, nelle stesse posizioni (la lista targeT deve essere lunga almeno quanto quella sorgente)

`static <T> void fill(List<? super T> l, T value)`

Riempie la lista con lo stesso valore.

`static <T> boolean addAll(Collection<? super T> c, T... values)`

Aggiunge tutti i valori ad una data collezione (rest. True se la collezione risultante cambia)

`static <T> boolean replaceAll(List<T> l, T oldValue, T newValue)` **1.4**

Restituisce tutti i valori di una collezione.

`static int indexOfSubList(List<?> l, List<?> s)` **1.4**

`static int lastIndexOfSubList(List<?> l, List<?> s)` **1.4**

`static void swap(List<?> l, int i, int j)` **1.4**

`static void reverse(List<?> l)`

Inverte l'ordine delle Liste.

`static void rotate(List<?> l, int d)` **1.4**

Ruota gli elementi di d posti (da sx a dx)

`static int frequency(Collection<?> c, Object o)` **5.0**

Restituisce il numero di oggetti uguali ad uno dato

`boolean disjoint(Collection<?> c1, Collection<?> c2)` **5.0**

Restituisce true se le collection argomento non hanno elementi in comune.

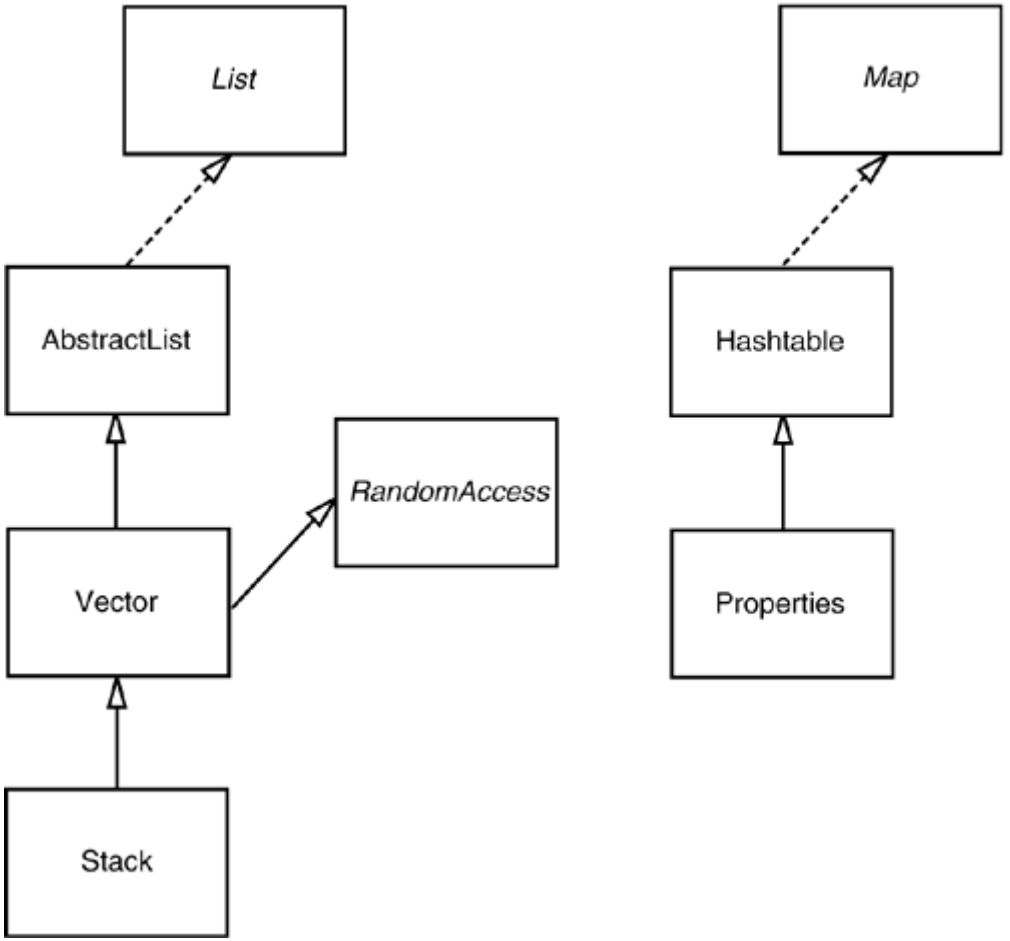
- Hashtable

  E' essenzialmente equivalente ad HashSet, ma i suoi metodi sono *Syncronised*

- La interfaccia Enumeration è equivalente ad Iterator, con due metodi hasMoreElement (hasNext) e nextElement (next). Il metodo statico Collections.enumeration offre una interfaccia enumeration per la compatibilità con codice precedente

Le classi container precedenti alla JFC

- Dalla versione Java 1.0 erano presenti delle calssi container Vector,Stack,HashTable e Properties che sono state integrate nella JFC



Properties

- Properties è una Map (Property Set) di tipo speciale dove chiavi e proprietà sono di tipo stringa. Inoltre è implementano un meccanismo di salvataggio, lettura su file delle proprietà, ed è molto usata per gestire file di configurazione. Consente di utilizzare una tabella di default (in fase di costruzione)

`Properties()`

`Properties(Properties defaults)`

Costruttori con o senza valori di default.

`String getProperty(String key)`

Restituisce la proprietà associata con la chiave, o se non è presente, la proprità conservata nella tabella di default, oppure null.

`String getProperty(String key, String defaultValue)`

Restituisce la proprietà associata ad una chiave, oppure un valore di default

`void load(InputStream in)`

Carica una lista di proprietà

`void store(OutputStream out, String commentString)`

Salva una lista di proprietà

La classe Stack estende Vector, ed ha due operazioni fondamentali, push e pop.

java.util.Stack<E> 1.0

- `E push(E item)`

  push `item` E sullo stack e restituisce `item`.

- `E pop()`

  pop l'item E che si trova in cima dallo stack (lo restituisce e lo toglie)

- `E peek()`

  Restituisce l'elemento in cima allo stack senza fare pop

BitSet

E' una classe che conserva una sequenza di bit, utile a rappresentare efficientemte dei parametri di configurazione di tipo FLAG.

```
get(i) //== valore dell'i-mo bit
```

```
set(i) //== imposta i-mo bit
```

```
clear(i) //== pone a 0 l'i-mo bit
```

Il metodo toString mostra le posizioni dei bit non 0, col costruttore si definisce la dimensione del bit set

	Map	Set	List	Ordine degli elementi	Elementi ordinati
HashMap	X			no	no
Hashtable	X			no	no
TreeMap	X			si	Con metodo di confronto esterno o con il confronto naturale
LinkedHashMap	X			Ordine di inserimento o ultimo accesso	no
HashSet		X		no	no
TreeSet		X		si	Con metodo di confronto esterno o con il confronto naturale
LinkedHashSet		X		Ordine inserimento	no
ArrayList			X	posizione	no
Vector			X	posizione	no
LinkedList			X	posizione	no
PriorityQueue				si	Ordinamento “to do”