Collezioni in Java

Emilio Di Giacomo e Walter Didimo

Strutture dati

- Una struttura dati è un “contenitore” in cui i dati sono organizzati in maniera che possano essere recuperati e manipolati efficientemente
- Un esempio di struttura dati che abbiamo utilizzato ampiamente è l’array
- In un array vengono memorizzati *n* elementi di uno stesso tipo
  - ogni elemento viene messo in corrispondenza con un indice da 0 a *n*-1
  - è possibile recuperare un elemento o modificarlo specificando il suo indice

Strutture dati statiche e dinamiche

- Una distinzione relativa alle strutture dati è la distinzione tra struttura dati statica e struttura dati dinamica
- In una struttura dati statica la capacità, cioè il numero di elementi che è possibile memorizzare nella struttura, è fissato all'atto della creazione
  - Un esempio di struttura dati statica è l'array
- In una struttura dati dinamica invece la capacità non è fissata a priori

Strutture dati statiche e dinamiche

- Una struttura dati statica ci obbliga a sapere il numero massimo di elementi che dovremo memorizzare all'interno della struttura nel momento in cui questa viene creata
- Ci sono casi in cui questa informazione non è nota e si vorrebbe poter disporre di una struttura dati in grado di crescere in base alle esigenze
- Abbiamo già parlato di strutture dati dinamiche in C

Strutture dati astratte e concrete

- Un'altra distinzione è quella tra strutture dati astratte e strutture dati concrete
- Una struttura dati astratta specifica il tipo di operazioni che è possibile eseguire sui dati, astraendo dalla specifica realizzazione
- Una struttura dati concreta è al contrario una realizzazione di una struttura dati astratta

Strutture dati astratte e concrete

- Ad esempio la struttura dati astratta pila (che abbiamo già incontrato) definisce due operazioni:
  - l'operazione *push(obj)* che inserisce l'oggetto obj nella pila
  - l'operazione *pop()* che rimuove dalla pila l'ultimo elemento inserito (tra quelli presenti) e lo restituisce
- Come si vede, la definizione di una struttura dati astratta non dice nulla su come debba essere realizzata ma soltanto quali operazioni è possibile eseguire su di essa

Strutture dati astratte e concrete

- Un possibile modo di realizzare una pila prevede di utilizzare un array ed una variabile intera *count* che indica la posizione successiva all'elemento in cima alla pila
  - Inizialmente *count* vale 0
- l'operazione *push(obj)* inserisce *obj* in posizione *count* ed incrementa *count*
- l'operazione *pop()* decrementa *count* e restituisce l'elemento che si trova in posizione *count* dopo il decremento

Strutture dati astratte e concrete

- Quella descritta è una possibile struttura dati concreta che realizza la struttura dati astratta pila
- Ovviamente non è l'unica possibile

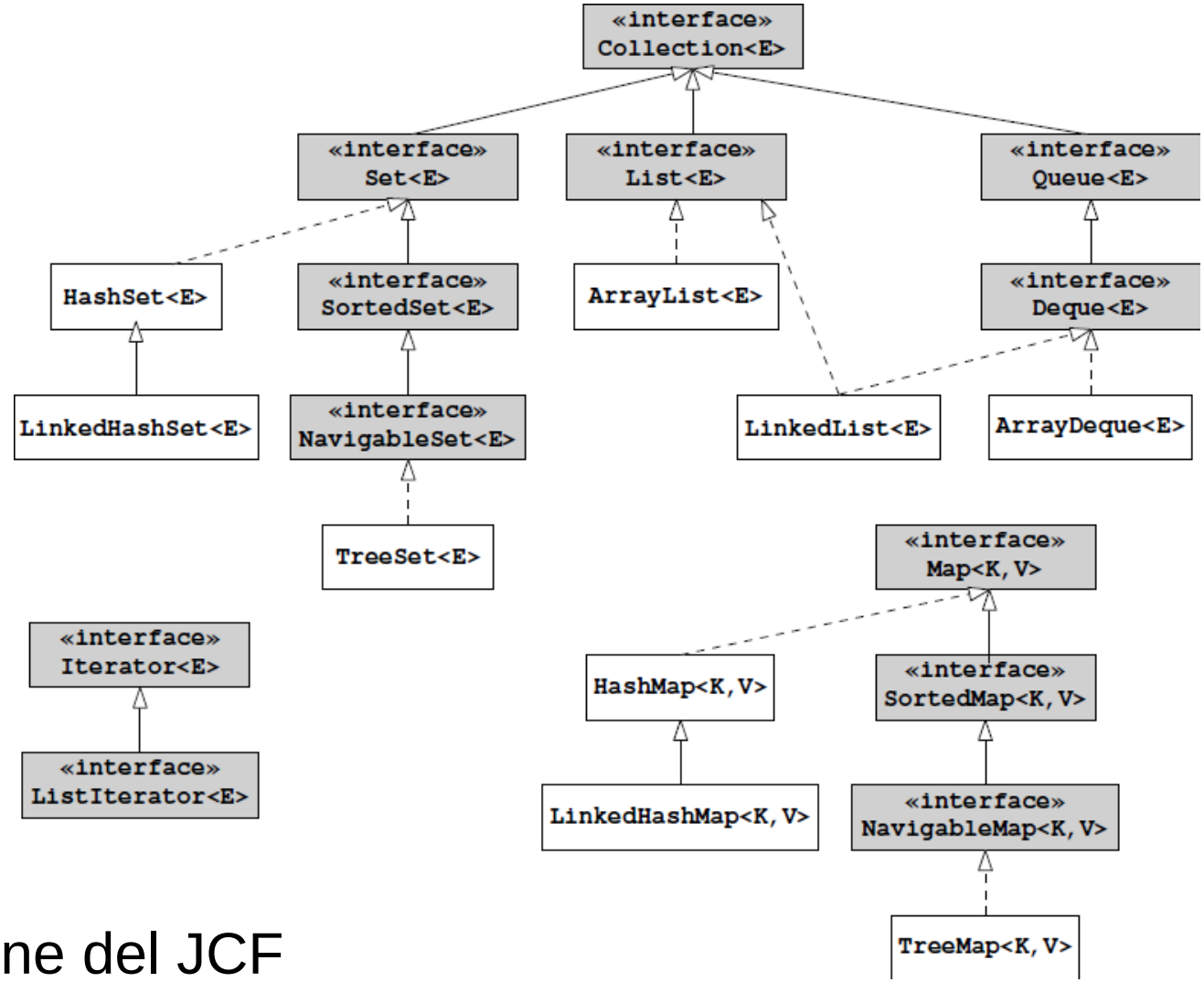
Strutture dati astratte e concrete

- Tipicamente per realizzare delle strutture dati si definisce il comportamento previsto dalla struttura dati astratta tramite una interface
- Si creano una o più strutture dati concrete che realizzano la struttura dati astratta tramite delle classi che implementano l'interface

Java Collection Framework

- Nell'API Java esistono una serie di interface e classi che definiscono e realizzano diversi tipi di strutture dati dinamiche, chiamate genericamente collezioni
- L'insieme di tali classi/interface viene chiamato Java Collection Framework (JCF)
- Le classi/interface del JCF si trovano nel package *java.util*

Java Collection Framework



Porzione del JCF

Collezioni Java

- Nel seguito vedremo alcune delle interfacce e classi Java per gestire le collezioni
- Non vedremo tutti i dettagli ma solo gli aspetti essenziali per poter utilizzare il JFC
- Prima però abbiamo bisogno di introdurre un ulteriore costrutto Java: i tipi generici (generics)

Tipi generici

- Un tipo generico (generics) è un tipo riferimento (dichiarato quindi per mezzo di una classe) definito parametricamente rispetto ad un altro tipo riferimento, chiamato tipo parametro
- Quando si usa un tipo generico è necessario associare al tipo parametro un tipo effettivo chiamato tipo argomento

Tipi generici

- Illustriamo i tipi generici con un esempio
  - non descriveremo tutti i dettagli dei tipi generici
- Supponiamo di voler definire una classe *Coppia* le cui istanze rappresentano coppie di elementi di uno stesso tipo (ad esempio coppie di interi, coppie di stringhe, ecc.)

- Una possibilità che abbiamo già visto nella lezione sull'ereditarietà prevede l'utilizzo della classe *Object*

La classe Coppia

```
public class Coppia{
   private Object el1;
   private Object el2;
   public Coppia(Object el1, Object el2){
       this.el1 = el1;
       this.el2 = el2;
   }
   public Object primoElemento(){
       return this.el1;
   }
   public Object secondoElemento(){
       return this.el2;
   }
}
```

Limiti della soluzione vista

- La soluzione precedente presenta alcuni limiti
- Innanzi tutto non c'è nessun modo di garantire che i due elementi memorizzati in un oggetto istanza della classe coppia siano dello stesso tipo:

  Coppia c=new Coppia("ciao", new Integer(1));

- Inoltre sono necessarie conversioni di tipo:

  String s=(String)c.primoElemento();

  Integer i=(Integer)c.secondoElemento();

Limiti della soluzione vista

- Ci piacerebbe che la classe *Coppia* sia definita in maniera generica (rispetto al tipo degli elementi) per permettere che gli elementi siano di qualunque tipo
- Vorremmo però che ad ogni utilizzo fosse possibile indicare il tipo da adoperare in quella occasione
- È possibile fare ciò con i tipi generici
- Vediamo la classe *Coppia* realizzata come classe generica

Tipi generici: esempio

```
public class Coppia<E> {

   private E el1;
   private E el2;

   public Coppia(E el1, E el2){
        this.el1=el1;
        this.el2=el2;
   }

   public E primoElemento(){ return el1; }

   public E secondoElemento(){ return el2; }

}
```

Tipi generici: commenti

- *Coppia<E>* è il tipo generico (si legge Coppia di *E*) mentre *E* è il tipo parametro
- In pratica stiamo definendo la classe *Coppia* parametricamente rispetto al tipo *E*:
  - assegnando ad *E* un tipo effettivo, ad esempio *String*, la classe diviene un tipo concreto (nell'esempio una coppia di stringhe)
  - il tipo assegnato ad E viene detto tipo argomento mentre il tipo concreto che si ottiene viene detto tipo parametrizzato

Tipi generici: terminologia

public class Coppia <E>{...}

Coppia<String> c=...

- *E*: tipo parametro
- *String*: tipo argomento
- *Coppia <E>*: tipo generico
- *Coppia<String>*: tipo parametrizzato

Tipi generici: commenti

- Si osservi come, nella definizione della classe *Coppia<E>*, il tipo *E* sia utilizzato in tutti i punti in cui è necessario far riferimento al tipo degli elementi che costituiscono la coppia:
  - ad esempio i parametri del costruttore sono di tipo *E*
  - il tipo restituito dai due metodi è il tipo *E*
- Associando ad *E* un tipo argomento, ad es. *String*:
  - i parametri del costruttore dovranno essere di tipo *String*
  - i valori restituiti dai due metodi saranno di tipo *String*

Uso di tipi generici: esempio

- Esempio di uso di una coppia di stringhe:

```
Coppia<String> c=new Coppia<String>("alfa","beta");
String s1=c.primoElemento();
String s2=c.secondoElemento();
```

- Esempio di uso di una coppia di interi (*Integer*):

```
Coppia<Integer> c1=new Coppia<Integer>(
                    new Integer(10),new Integer(20));
int i1=c1.primoElemento().intValue();
int i2=c1.secondoElemento().intValue();
```

Ancora sui tipi generici

- Nel definire una classe (o interface) generica è possibile utilizzare più tipi argomento
- Se ad esempio volessimo definire una coppia i cui elementi possono essere di tipo diverso potremmo definire la classe *Coppia<E,T>*
- In questo caso potremmo poi dar luogo, ad esempio, ai seguenti tipi parametrizzati:
  - *Coppia<String, Integer>*
  - *Coppia<String, String>*
  - *Coppia <Double, Short>*

Ancora sui tipi generici

- Nello scrivere il corpo di una classe generica il tipo (o i tipi) argomento va utilizzato come se fosse un tipo concreto:
  - può essere usato, ad esempio, come tipo delle variabili, come tipo di ritorno dei metodi, come tipo dei parametri
- Ci sono però delle restrizioni

Ancora sui tipi generici

- Non è possibile usare un tipo argomento per:
  - definire un campo statico
  - creare un array
  - istanziare oggetti

private static E var; // ERRORE!!

E[] arr = new E[n]; // ERRORE!!

E var = new E(); // ERRORE!!

Tipi parametro vincolati

- Nell'esempio visto della classe *Coppia*, il tipo parametro *E* poteva essere rimpiazzato da qualunque tipo (esclusi i tipi primitivi)
- È possibile restringere i tipi che possono essere usati come tipi argomento
- Ad es., supponiamo di voler creare una classe le cui istanze sono coppie di numeri dotata di un metodo che somma i valori dei due elementi della coppia

Tipi parametro vincolati

```
public class CoppiaNumerica<E extends Number>{
	private E el1;
	private E el2;
	...
	public double somma(){
		return el1.doubleValue()+el2.doubleValue();
	}
}
```

- In questo caso si potranno usare come tipo argomento soltanto la classe *Number* o sue sottoclassi
- ciò garantisce che *el1* ed *el2* siano dotati del metodo *doubleValue()*

Uso dei tipi parametrizzati

- Un tipo parametrizzato può essere usato come un qualunque tipo "normale"
- Non è possibile però definire e creare array:

Coppia<String>[] arr=new Coppia<String>() // ERRORE!!

- Bisogna inoltre capire bene il modo in cui i tipi parametrizzati vengono gestiti per usarli correttamente

Uso dei tipi parametrizzati

- Supponiamo di voler scrivere un metodo che riceve come parametro una coppia di numeri e restituisce la loro somma
- Potremmo pensare di scrivere:

```
public static double somma(Coppia<Number> c){
      Number n1=c.primoElemento();
      Number n2=c.secondoElemento();
      return n1.doubleValue()+n2.doubleValue();
}
```

Uso dei tipi parametrizzati

- Ci si potrebbe aspettare che con il codice precedente sia possibile scrivere:

```
Coppia <Integer> c=
      new Coppia(new Integer(1), new Integer(2));
somma(c);
```

- L'invocazione *somma(c)* invece darà errore
- Il motivo è che, benché *Integer* sia una sottoclasse di *Number*, *Coppia<Integer>* non è una sottoclasse di *Coppia<Number>*

Segnaposto

- Come possiamo scrivere il metodo *somma*?
- Possiamo usare i segnaposto (wildcard)
- Un segnaposto è indicato dal carattere *?* e indica un tipo argomento non specificato

```
public static double somma(Coppia<? extends Number> c){
        Number n1=c.primoElemento();
        Number n2=c.secondoElemento();
        return n1.doubleValue()+n2.doubleValue();
}
```

Segnaposto

- Nell'esempio precedente il segnaposto indica che *c* dovrà essere una coppia i cui elementi sono un sotto-tipo di *Number* (incluso *Number* stesso)
  - in questo caso si parla di segnaposto vincolato
- *Number* è un vincolo superiore al tipo del segnaposto

Segnaposto

- È possibile imporre un vincolo inferiore al segnaposto:
  - *<? super Number>* indica che il tipo argomento deve essere *Number* o un suo supertipo
- È possibile usare il segnaposto non vincolato *<?>*
  - ad es. se definissimo un parametro *p* di tipo *Coppia<?>,* potremmo assegnare a *p* un oggetto di tipo *Coppia<String>* ma anche *Coppia<Object>* o *Coppia<Integer>*

Metodi generici

- È possibile utilizzare i tipi parametro anche nella definizione di metodi (o costruttori) sia all'interno di classi generiche che di classi "normali"
- Supponiamo ad esempio di voler scrivere un metodo che date due coppie ci dice se sono uguali

Metodi generici

- Potremmo pensare di scrivere

```
public static boolean uguali(Coppia<?> c1, Coppia<?> c2){
  boolean b1,b2;
  b1 = c1.primoElemento().equals(c2.primoElemento());
  b2 = c1.secondoElemento().equals(c2.secondoElemento());
  return b1 && b2;
}
```

Metodi generici

- Con la definizione precedente però sarebbe possibile passare al metodo due coppie di tipi parametrizzati diversi

```
Coppia<String> c1 = new Coppia<String>("A","B");
Coppia<Integer> c2 = new Coppia<Integer>(2,4);
uguali(c1,c2)
```

- Noi vorremmo che i tipi parametrizzati coincidessero

Metodi generici

- Possiamo riscrivere il codice come segue

```
public static <T> boolean uguali(Coppia<T> c1, Coppia<T> c2){
  boolean b1,b2;
  b1 = c1.primoElemento().equals(c2.primoElemento());
  b2 = c1.secondoElemento().equals(c2.secondoElemento());
  return b1 && b2;
}
```

- *<T>* è una dichiarazione tipo parametro analoga a quella che segue il nome di una classe generica
- Essa introduce un tipo *T* non specificato che può essere utilizzato nella scrittura del metodo

Invocazione di metodi generici

- All'atto dell'invocazione di un metodo generico è necessario associare al tipo parametro un tipo argomento
  - nel seguente codice si assume che *uguali* sia definito nella classe *Prova*

```
Coppia<String> c1 = new Coppia<String>("A","B");
Coppia<String> c2 = new Coppia<String>("C","D");
Prova.<String>uguali(c1,c2);
```

Invocazione di metodi generici

- In realtà, il tipo argomento può essere dedotto dal tipo degli oggetti passati come parametro
- Pertanto il seguente codice è equivalente al precedente

```
Coppia<String> c1 = new Coppia<String>("A","B");
Coppia<String> c2 = new Coppia<String>("C","D");
Prova.uguali(c1,c2);
```

Metodi generici: esempio 2

- Il seguente metodo cerca un elemento in un array
- Definendo l'array come *T[]* si fa in modo che il metodo possa funzionare con array di qualunque tipo
- Si garantisce inoltre che l'elemento da cercare sia di tipo *T*

```
public static <T> boolean contiene(T[] a, T elem){
	boolean trovato = false;
	for (int i = 0; i<a.length; i++){
		if (a[i].equals(elem))
			trovato = true;
	}
	return trovato;
}
```

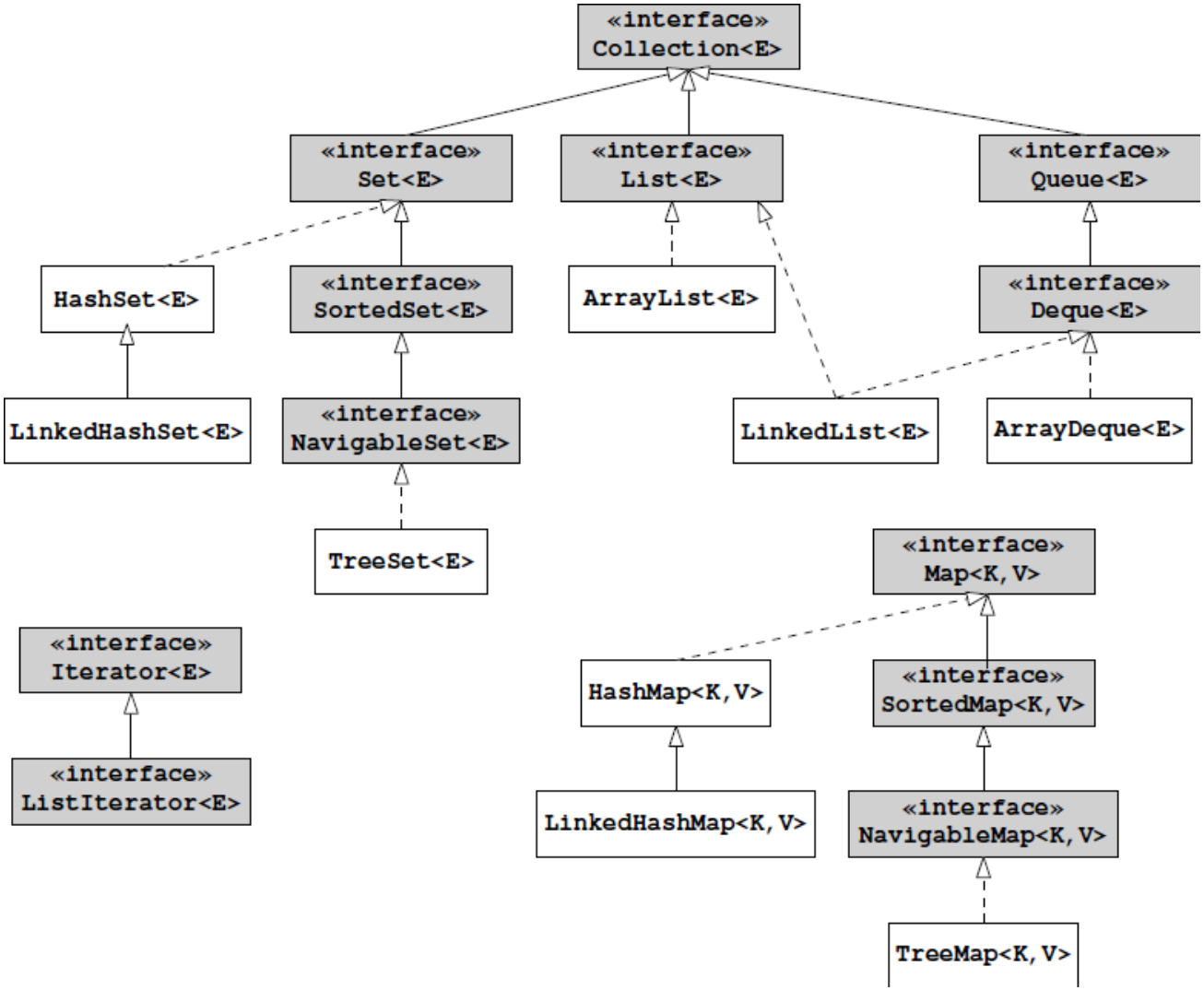
Metodi generici: esempio 3

- Il seguente metodo restituisce il massimo elemento in un array
- Definendo l'array come *T[]* si fa in modo che il metodo possa funzionare con array di qualunque tipo
- Si garantisce inoltre che il valore restituito sia di tipo *T*
- Il tipo parametro vincolato garantisce che l'array sia di un tipo numerico

```
public static <T extends Number> T massimo(T[] a){
        T massimo = a[0];
        for (int i = 1; i<a.length; i++){
                if (a[i].doubleValue()>massimo.doubleValue())
                        massimo = a[i];
        }
        return massimo;
}
```

Java Collection Framework

Java Collection Framework



Java Collection Framework

- Nel seguito descriveremo le interfacce e classi principali del JFC
- Non vedremo tutti i dettagli per i quali si rimanda alla documentazione ufficiale Java

Interface Collection<E>

- L'interface *Collection<E>* rappresenta genericamente una collezione di oggetti, chiamati elementi
- È il super-tipo di quasi tutte le strutture di dati astratte del JCF
  - fa eccezione *Map<K,V>*

Interface Collection<E>

- Alcuni metodi dell'interface *Collection<E>* sono i seguenti
- *int size()* – restituisce il numero di elementi che essa contiene
- *boolean isEmpty()* – restituisce *true* se la collezione non contiene alcun elemento
- *boolean contains(Object o)* – restituisce *true* se la collezione contiene *o*
- *boolean add(E e)* – aggiunge l'oggetto *e* (a meno che la collezione escluda duplicati e *e* sia già presente) – restituisce *true* se *e* è stato aggiunto

Interface Collection<E>

- *boolean remove(Object o)* – rimuove dalla collezione una singola istanza di *o*, restituendo *true* nel caso in cui la rimozione sia avvenuta (se un oggetto uguale ad *o* non esiste restituisce *false*)
- *Iterator iterator()* – restituisce un iteratore che permette di visitare in sequenza tutti gli elementi della collezione (v. di seguito)
- *void clear()* – rimuove tutti gli elementi della collezione

Interface Collection<E>

- In tutti i metodi che devono verificare l'uguaglianza tra elementi (cioè *contains* e *remove*), l'uguaglianza viene stabilita sulla base del metodo *equals*, quindi:
  - il metodo *contains(Object o)* restituisce *true* se la collezione contiene un elemento *e* tale che *e.equals(o)* è *true*
  - il metodo *boolean remove(Object o)* rimuove dalla collezione un elemento *e* tale che *e.equals(o)* è *true*
- Ciò è vero in generale per tutte le classi/interface del JCF

Interface Iterator<E>

- Per scandire gli elementi di una struttura dati si possono utilizzare gli iteratori
- Un iteratore è un oggetto che permette di visitare in sequenza tutti gli oggetti contenuti in una struttura dati
- Un iteratore tiene traccia in ogni momento dell'ultimo elemento visitato e offre (almeno) due operazioni:
  - una per verificare se ci sono ulteriori elementi
  - una per avanzare nella visita

Interface Iterator<E>

- I metodi dell'interface *Iterator<E>* sono i seguenti
- *boolean hasNext()* – restituisce *true* se ci sono ancora elementi da visitare
- *E next()* – restituisce il prossimo elemento da visitare e avanza nella visita
- *void remove()* – rimuove dalla collezione l'ultimo elemento restituito

Uso di iteratori - esempio

- Vediamo come viene usato un iteratore

```
...
Collection<String> c;
String elem;
   ... crea e popola la collezione c ...
/* visita c usando un iteratore */
Iterator<String> i = c.iterator();
while (i.hasNext()) {
    /* accede al prossimo elemento ... */
    elem = i.next();
    /* ... e lo elabora */
    ...
}
```

Set<E>, List<E> e Queue<E>

- L'interface *Collection<E>* viene estesa da alcune interface che definiscono diverse strutture dati astratte:
  - *Set<E>*
  - *List<E>*
  - *Queue<E>*

L'interface Set<E>

- L'interface *Set<E>* definisce la struttura dati insieme (in senso matematico):
- Un insieme è una collezione di elementi in cui non ci sono duplicati
- *Set<E>* non aggiunge alcun metodo a quelli definiti da *Collection<E>* ma ridefinisce la semantica dei metodi di aggiunta (ad es. *add(...)*) specificando che un elemento non deve essere aggiunto se già presente

L'interface List<E>

- L'interface *List<E>* definisce la struttura dati lista
- Una lista è una sequenza di elementi:
  - possono esserci dei duplicati
  - ogni elemento ha una posizione nella sequenza

Interface List<E>

- Poiché *List* estende *Collection* ne erediterà tutti i metodi; aggiunge altri metodi utili per la gestione di una lista, tra cui:
- *E get(int index)* – restituisce l'elemento di posizione *index* all'interno della lista
- *E set(int index, E elem)* – sostituisce con *elem* l'elemento di posto *index* della lista, restituendo il contenuto precedente
- *int indexOf(Object o)* – restituisce la posizione del primo elemento della lista uguale ad *o*. Restituisce *-1* se un tale elemento non esiste

Interface List<E>

- *void add(int index, E elem)* – aggiunge alla lista nella posizione *index* l'elemento *elem*, spostando tutti gli elementi successivi a destra di una posizione
- *E remove(int index)* – rimuove e restituisce l'elemento di posizione *index* della lista, spostando tutti gli elementi successivi a sinistra di una posizione

L'interface Queue<E>

- L'interface *Queue<E>* definisce la struttura dati coda
- Una coda è una collezione di elementi che vengono estratti l'uno dopo l'altro secondo un ordine preciso
  - in ogni momento esiste un elemento che è il prossimo che sarà estratto
  - tale elemento è detto testa della coda

L'interface Queue<E>

- La politica con cui viene stabilito l'ordine di estrazione non è definita dall'interface *Queue<E>*
- L e classi che implementano questa interface possono seguire politiche diverse
- Generalmente un coda viene gestita con politica FIFO (First In First Out)
  - il primo arrivato è il primo ad essere estratto
- Possono essere usati altri criteri
  - Ad esempio nelle code di priorità il prossimo elemento da estrarre viene stabilito in base ad un valore di priorità associato ad ogni elemento presente nella coda

L'interface Queue<E>

- *Queue* (ri)definisce metodi per aggiungere, rimuovere o esaminare l'elemento in testa
- Ogni metodo è disponibile in due forme
  - una dà errore se l'operazione non è possibile
  - l'altra restituisce un opportuno valore se l'operazione non è possibile
- Aggiunta:
  - *boolean add(E e)* – aggiunge l'oggetto *e* alla coda. Restituisce sempre *true*. Se l'elemento non può essere aggiunto perché la coda ha capacità limitata genera un errore
  - *boolean offer(E e)* – aggiunge l'oggetto *e* alla coda. Restituisce *true* se *e* è stato aggiunto. Se l'elemento non può essere aggiunto perché la coda ha capacità limitata restituisce *false*
- Rimozione:
  - *E remove()* – rimuove e restituisce l'elemento di testa. Se la coda è vuota genera un errore
  - *E poll()* – rimuove e restituisce l'elemento di testa. Se la coda è vuota restituisce *null*

L'interface Queue<E>

- Consultazione:
  - *E element()* – restituisce l'elemento in testa. Se la coda è vuota genera un errore
  - *E peek()* – restituisce l'elemento in testa. Se la coda è vuota restituisce *null*

L'interface Deque<E>

- L'interface *Queue<E>* è estesa dall'interface *Deque<E>* che rappresenta la struttura dati coda doppia (double ended queue)
- Una coda doppia permette di operare sull'elemento di testa e su quello di coda
- Per ognuno dei metodi di *Queue<E>*, *Deque<E>* offre due versioni una che opera sulla testa e una sulla coda
  - *add(E e)* → *addFirst(E e)*, *addLast(E e)*
  - *offer(E e)* → *offerFirst(E e)*, *offerLast(E e)*
  - ...

L'interface Map<K,V>

- L'interface *Map<K,V>* definisce la struttura dati mappa
- Una mappa memorizza coppie *⟨chiave,valore⟩* e permette di recuperare efficientemente un valore tramite la corrispondente chiave
  - Ad esempio è possibile utilizzare una mappa per memorizzare degli oggetti di tipo *Persona* (valori) usando come chiave il codice fiscale della persona
  - In questo modo è possibile recuperare efficientemente una persona dato il suo codice fiscale

L'interface Map<K,V>

- Alcuni metodi definiti nell'interface *Map<K,V>* sono i seguenti
- *void clear()* - Rimuove tutte le coppie dalla mappa.
- *boolean containsKey(Object key)* - Restituisce *true* se la mappa contiene una coppia con chiave *key*
- *boolean containsValue(Object value)* - Restituisce *true* se la mappa contiene una o più coppie il cui valore sia value
- *V get(Object key)* - Restituisce il valore associato alla chiave *key*, o *null* se non esiste alcuna coppia la cui chiave è *key*
- *boolean isEmpty()* - Restituisce *true* se la mappa è vuota

L'interface Map<K,V>

- *Set<K> keySet()* - Restituisce l'insieme delle chiavi contenute nella mappa sotto forma di un oggetto di tipo *Set*
- *Set<Map.Entry<K,V>> entrySet()* - Restituisce l'insieme delle coppie contenute nella mappa sotto forma di un oggetto di tipo *Set*. *Map.Entry<K,V>* è una interface definita all'interno dell'interface *Map<K,V>*. Un'istanza di *Map.Entry<K,V>* rappresenta una coppia ⟨*chiave*, *valore*⟩ che è possibile memorizzare in una mappa.
- *V put(K key, V value)* - Inserisce la coppia ⟨*key*, *value*⟩ nella mappa. restituisce il valore precedentemente associato con la chiave (o *null*)
- *V remove(Object key)* - Rimuove la coppia con chiave *key* dalla mappa, se presente. Restituisce il valore precedentemente associato con la chiave (o *null*)
- *int size()* - Restituisce il numero di coppie contenute nella mappa.
- *Collection<V> values()* - Restituisce i valori contenuti nella mappa sotto forma di un oggetto di tipo *Collection*

Altre interface

- Oltre quelle viste esistono delle interface che non definiscono strutture dati diverse da quelle viste ma le estendono per aggiungere funzionalità utili
- Ad esempio, *Set<E>* è estesa dall'interface *SortedSet<E>*
- In un insieme di tipo *SortedSet<E>* gli elementi vengono mantenuti ordinati ed è possibile estrarre degli elementi in base a criteri che tengono conto dell'ordinamento
- È possibile, ad esempio, estrarre da un insieme di tipo *SortedSet<E>* l'elemento più piccolo oppure estrarre il sottoinsieme formato dagli elementi minori di un elemento dato
- Inoltre quando si usa un iteratore per scandire gli elementi di un *SortedSet<E>*, essi vengono restituiti ordinati

Altre interface

- L'interface *NavigableSet<E>* estende *SortedSet<E>* aggiungendo ulteriori operazioni
  - Ad esempio si può cercare l'elemento più vicino ad uno dato, oppure scandire gli elementi in ordine inverso
  - Per maggiori dettagli si rimanda alla documentazione Java
- Una situazione analoga si ha per l'interface *Map<K,V>* che è estesa da *SortedMap<K,V>* a sua volta estesa da *NavigableMap<K,V>*
  - In una mappa di tipo *SortedMap<K,V>* o *NavigableMap<K,V>* le coppie memorizzate nella mappa vengono ordinate in base ai valori delle chiavi

Implementazioni

- Per ognuna delle strutture dati astratte che abbiamo visto esistono classi che forniscono diverse implementazioni
- Alcune di queste sono implementazioni general purpose, cioè implementazioni di uso generale
  - altre sono implementazioni specifiche per situazioni particolari

Implementazioni

- La tabella seguente mostra le combinazioni esistenti per le implementazioni general purpose

	Tabelle Hash	Array Dinamici	Alberi Bilanciati	Liste Collegate	Tabelle Hash + Liste Collegate
Set	HashSet		TreeSet*		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap**		LinkedHashMap
* implementa NavigableSet					
** implementa NavigableMap					

Collezioni: esempio d'uso

- Vediamo ora un esempio di uso di alcune delle strutture dati che abbiamo visto
- Supponiamo di voler scrivere un programma che chiede all'utente di inserire un testo (cioè una sequenza di parole) e poi mostra all'utente:
  - la sequenza di parole inserita
  - le parole distinte presenti nella sequenza
  - le parole distinte presenti nella sequenza in ordine alfabetico
  - la frequenza di ciascuna parola

Collezioni: esempio d'uso

- Risolviamo il problema inserendo le parole inserite dall'utente in quattro diverse collezioni
- Per memorizzare la sequenza delle parole inserite (nell'ordine di inserimento) usiamo una lista
- Per eliminare i duplicati usiamo un insieme
- Per ordinare alfabeticamente le parole distinte usiamo un insieme ordinato
- Per contare le frequenze usiamo una mappa, le cui chiavi sono le parole i cui valori sono le frequenze

Collezioni: esempio d'uso

- Per stampare il contenuto delle varie collezioni si utilizza il metodo *stampa(...)* che riceve in ingresso una collezione e ne stampa il contenuto utilizzando un iteratore
- Si noti che tale metodo viene utilizzato anche per stampare il contenuto della mappa; a questo scopo si usa il metodo *entrySet()* per ottenere le coppie contenute nella mappa sotto forma di insieme

La classe ProvaCollezioni

```
import fond.io.*;
import java.util.*;

public class ProvaCollezioni{

      /* stampa gli elementi della collezione c */
      private static void stampa(Collection<?> c){
            Iterator<?> it = c.iterator();
            while (it.hasNext())
                  System.out.print(it.next()+" ");
            System.out.println();
      }
      ...

}
```

La classe ProvaCollezioni

```
public class ProvaCollezioni{
        ...

        public static void main(String[] args){
                InputWindow in = new InputWindow();

                /* crea alcune collezioni di diverso tipo */
                List<String> list = new ArrayList<String>();
                Set<String> set = new HashSet<String>();
                SortedSet<String> sset = new TreeSet<String>();
                Map<String,Integer> map = new HashMap<String, Integer>();

                /* fa inserire all'utente una sequenza di parole */
                String s = in.readString("Inserisci una parola");
                while (s!=null){
                        list.add(s);
                        set.add(s);
                        sset.add(s);
                        int i = 0;
                        if (map.containsKey(s))
                                i = map.get(s);
                        map.put(s, i+1);      // incrementa la frequenza
                        s = in.readString("Inserisci una parola");
                }

                ...
        }
}
```

La classe ProvaCollezioni

```
public class ProvaCollezioni{

        ...

        public static void main(String[] args){
                ...
                System.out.println("Sequenza inserita");
                stampa(list); // parole in ordine di inserimento

                System.out.println("\nParole distinte");
                stampa(set); // parole distinte

                System.out.println("\nParole distinte ordinate");
                stampa(sset); // parole distinte in ordine alfabetico

                System.out.println("\nFrequenza di ogni parola");
                stampa(map.entrySet()); // frequenze delle parole
        }
}
```