Tipi di Dati Astratti (ADT)

Il tipo di dati astratto *Pila*

Le pile (**stack**) sono un tipo di dati astratto molto semplice, ma dal diffuso impiego.

Intuitivamente, una pila è un sequenza di zero o più elementi $\langle A_1\ A_2\ \ldots\ A_n \rangle$

in cui è possibile aggiungere o togliere elementi soltanto ad un estremo della sequenza detto **cima** (**top**) della pila.

Politica di accesso: **LIFO (Last In, First Out)**
[ultimo elemento inserito, primo elemento rimosso]

Esempi: una pila di piatti, un tubetto di pastiglie, ...

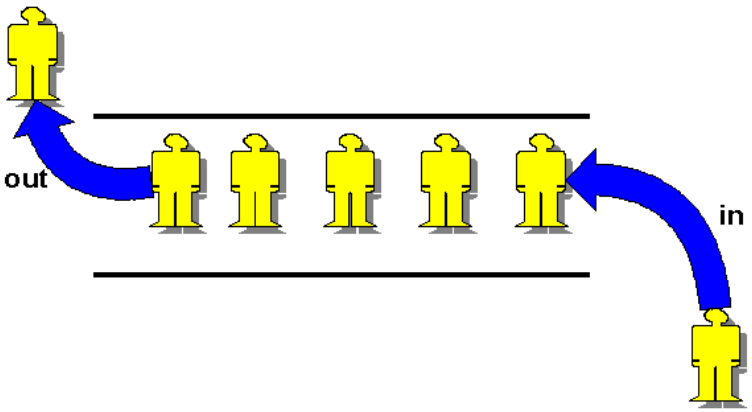
...una pila di libri...

Le **operazioni** definite su una pila sono le seguenti:

- **Inserimento** di un elemento *in cima* alla pila (normalmente chiamata **push**)
- **Estrazione** dell'elemento *in cima* alla pila (chiamata **pop**)
- **Lettura** dell'elemento *in cima* alla pila, senza eliminarlo (chiamata **top** o **peek**)
- **Controllo** se la pila è **vuota** oppure no (predicato **isEmpty**)
- **Svuotamento** della pila (chiamata **clear**)

Il tipo di dati astratto *Coda*

Una coda (**queue**) è un sequenza di zero o più elementi $\langle A_1\ A_2\ \ldots\ A_n \rangle$



in cui è possibile aggiungere elementi soltanto ad un estremo della sequenza, detto **fondo** (**back**), ed è possibile toglierli soltanto dall'altro estremo, detto **testa** (**front**).

Politica di accesso: **FIFO (First In, First Out)**
[primo elemento inserito, primo elemento rimosso]

Esempi: una coda di persone alla cassa di un cinema, una coda di ordini da evadere, una coda di file da stampare, ...

...una coda all'ingresso di un palazzo...

Le **operazioni** definite su una coda sono le seguenti :

- **Inserimento** di un elemento *in fondo* alla coda (chiamata **enqueue**)
- **Estrazione** dell'elemento *in testa* alla coda (chiamata **dequeue**)
- **Lettura** dell'elemento *in testa* alla coda, senza eliminarlo (chiamata **front**)
- **Controllo** se la coda è **vuota** oppure no (predicato **isEmpty**)
- **Svuotamento** della coda (chiamata **clear**)

Si noti che pile e code hanno essenzialmente le stesse operazioni, ma con significato (e quindi nome) diverso!

Il tipo di dati astratto *Lista*

Una lista è una struttura dati che permette di memorizzare dati in maniera sequenziale, è una collezione ordinata in cui si può accedere ad ogni elemento (non solo al primo o all'ultimo inserito, come nel caso di pile e code).

Una lista è una sequenza di zero o più elementi $\langle A_1\ A_2 \ldots A_n \rangle$

in cui è possibile aggiungere o rimuovere elementi in posizioni arbitrarie (sono permessi valori duplicati, a differenza di un insieme o *set: collezione non ordinata, senza duplicati*).

A differenza di pile e code, non vi è completo accordo sull'insieme di operazioni che caratterizzano le liste. Le **operazioni** fondamentali definite su una lista, (con riferimento alle classi[1] che implementano in Java l'interfaccia ***List*** che permette accesso indicizzato e ricerca) sono le seguenti:

- **Inserimento** di un elemento in fondo alla lista o in posizione indicizzata (chiamata **add**)
- **Rimozione** di un dato elemento dalla lista (chiamata **remove**)
- **Lettura** di un elemento(chiamata **get**) e modifica (chiamata **set**)
- **Verifica** se un dato elemento è presente nella lista (chiamata **contains**)
- **Controllo** se la lista è **vuota** oppure no (predicato **isEmpty**)
- **Svuotamento** della lista (chiamata **clear**)
- Creazione di un **iteratore** sulla lista (chiamata **iterator**)

Un **iteratore** consente di visitare in sequenza tutti gli elementi di strutture in cui gli elementi **non** sono tutti **accessibili in modo diretto** (ad esempio *liste*, *liste concatenate anche doppie*, *alberi binari*, ecc...).

In Java un'interfaccia **iterator** (e sue specializzazioni) permette di astrarre il problema di iterare su tutti gli elementi di una collection (un oggetto che rappresenta un insieme di oggetti).

[1] ArrayList e l'equivalente Vector (modificate con l'introduzione di ***Generics*** API Enhancements in J2SE 5.0), sono entrambe classi membro della Java Collections Framework

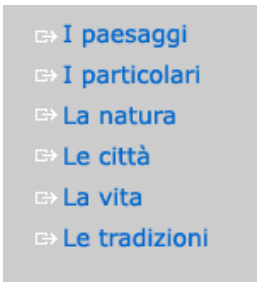
Metodi degli iteratori

Uso un iteratore per visitare una struttura dati:

- Si crea l'iteratore invocando il metodo **iterator()** sulla struttura dati

- Si usano i metodi forniti dall'iteratore per visitarne gli elementi. I metodi ed il loro significato sono descritti nell'interfaccia **Iterator**, dichiarata nella Java Collections API (package **java.util**):

```
public interface Iterator {
   boolean hasNext();
   Object next();
   void remove();
}
```



La API fornisce una precisa specifica dei metodi:

- **hasNext** restituisce **true** se esiste ancora un elemento della struttura dati da esaminare, **false** altrimenti;

- **next** restituisce il prossimo elemento da esaminare, e lancia un'eccezione di tipo **NoSuchElementException** se questo non esiste;

  Attenzione: **NoSuchElementException** è definita in **java.util** e va importata.

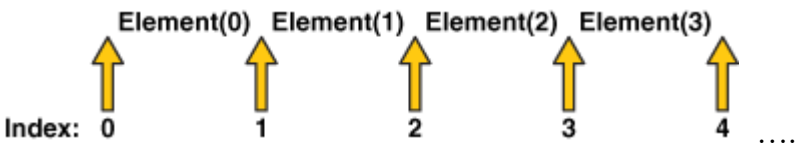
- **remove** cancella dalla struttura dati l'elemento restituito dall'ultima chiamata al metodo **next**, e può essere invocato una sola volta dopo ogni **next**. Se **remove** viene invocato appena l'iteratore è stato costruito oppure la seconda volta dopo una **next**, viene lanciata una **IllegalStateException**.

  Attenzione: **IllegalStateException** è definita in **java.lang** e quindi non è necessario importarla.

  Attenzione: questo metodo è **opzionale** (alcune strutture dati adottano iteratori senza possibilità di rimuovere elementi → in questi casi il metodo **remove** deve lanciare una un'eccezione di tipo **UnsupportedOperationException**).

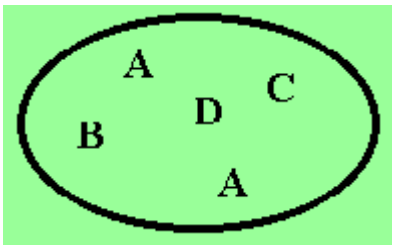
L'iteratore standard della classe **`Vector`** scandisce gli elementi dalla posizione **`0`** fino alla posizione **`size()-1`**.

Un iteratore specializzato di tipo **ListIterator** permette di scorrere una lista in entrambe le direzioni, con possibilità di modificare gli elementi (rimpiazzare ed inserire oltre a rimuovere) durante l'iterazione; la posizione corrente dell'iteratore sulla lista è sempre precedente o successiva `rispetto all'elemento:`

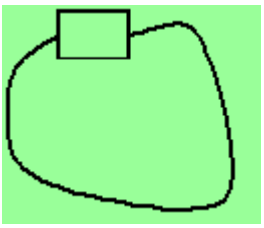


Un'interpretazione grafica di un iteratore

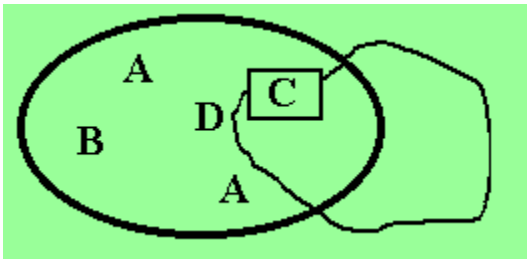
Una collezione
(non facciamo assunzioni
su ordine e ripetizioni dei suoi elementi)



Un iteratore per la collezione ottenuto con
`Iterator it = nomecoll.iterator();`

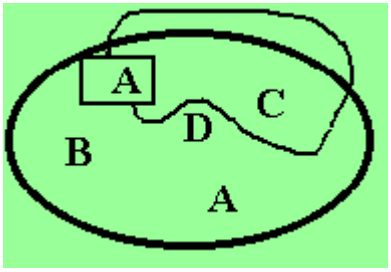


`it.next()` restituisce **`C`**
come effetto collaterale,
l'elemento restituito
viene marcato come
"esaminato".



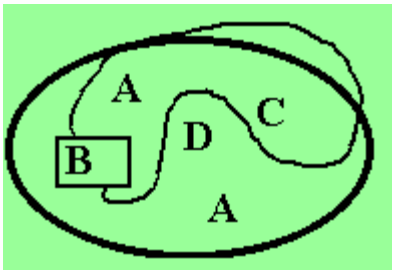
`it.next()` restituisce **`A`**

l'elemento **`C`** viene messo nel "sacchetto"
per non essere più considerato



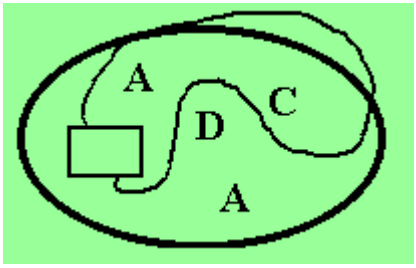
`it.next()` restituisce **`B`**

`it.hasNext()` restituisce **`true`**
perché c'è almeno un elemento
"fuori dal sacchetto"



`it.remove()` cancella dalla collezione **`B`**,
l'elemento nella finestra (l'ultimo visitato).

Un'altra invocazione di **`it.remove()`**
ora causerebbe un'eccezione,
perché la finestra è vuota.



Collezioni: strutture che raggruppano diversi dati, usualmente omogenei

Una *collection* — a volte chiamata **container** — è semplicemente un oggetto che raggruppa più elementi in una singola unità. Le *collections* sono usate per immagazzinare, recuperare, manipolare e trasmettere dati aggregati anche non omogenei. Solitamente rappresentano elementi che compongono un gruppo naturale come un mazzo da poker (una *collection* di carte) o un elenco telefonico (una *mappa* di nomi e numeri telefonici). Altri esempi: insiemi, multi-insiemi, sequenze, tabelle, matrici, pile, code, liste, alberi, ...

Esempi in Java: array, istanze di Vector, istanze di classi come ArrayList, LinkedList, TreeMap, HashMap ...

Un *array*, infatti, è una collezione con certe caratteristiche: un insieme ordinato di elementi ad accesso diretto tramite indice. Un **Vector** contiene una sequenza di dati (oggetti Java di tipo generico che evitano operazioni di casting) il cui numero può via via raddoppiare, e fornisce metodi per inserire, eliminare, cercare dati ed altro.

Le collezioni più importanti sono:

- *liste*, o *sequenze*, ordinate di elementi che possono essere inseriti e rimossi in ogni posizione;
- *insiemi* di elementi non replicati;
- *dizionari*, detti anche *hash tables*, che fanno corrispondere un elemento "indice" con un elemento "valore".

Una classe che rappresenta una collezione fornisce metodi per **inserire**, **rimuovere**, **trovare**, **ordinare**, ecc., i suoi elementi.

In generale, dunque, una collezione è caratterizzata da alcune proprietà strutturali e da un insieme di operazioni per accedere e manipolare i dati in esse contenuti.

Le collezioni sono utili anche in piccoli programmi, indispensabili in grandi applicazioni infatti, in genere, esistono due possibilità: o lo sviluppatore definisce le proprie classi che implementano le collezioni di cui ha bisogno per la propria applicazione, oppure utilizza alcune classi di libreria che realizzano vari tipi di collezione. Ovviamente la seconda soluzione è da preferire perché evita di inventare la ruota ogni volta.

Per una collezione è importante non solo il tipo di funzionalità offerte, ma anche il modo in cui le funzionalità sono implementate. Questo aspetto non deve essere trasparente allo sviluppatore che deve avere la possibilità di scegliere l'implementazione più adatta al proprio caso come compromesso tra benefici/controindicazioni.

Ad esempio, due liste, pur offrendo esattamente le stesse funzionalità, potrebbero essere implementate rispettivamente come un *array* e come *insieme di oggetti linkati*.

La prima implementazione usa la memoria in modo più efficiente e permette accessi rapidi ma è penalizzante se occorre inserire e rimuovere oggetti di frequente perché richiede riallocazioni dell'array. La seconda implementazione ha vantaggi e svantaggi opposti. Se venisse fornita un'unica classe lista che maschera completamente ogni aspetto implementativo, lo sviluppatore non avrebbe la possibilità di migliorare le prestazioni dell'applicazione.

Per non costringere, dunque, il programmatore a reinventarsi programmi per la definizione e gestione di collezioni, sono stati sviluppati diversi ***Collections Frameworks*** cioè architetture (software) unificate per rappresentare e manipolare collezioni.

Un **Collections Framework** tipicamente contiene:

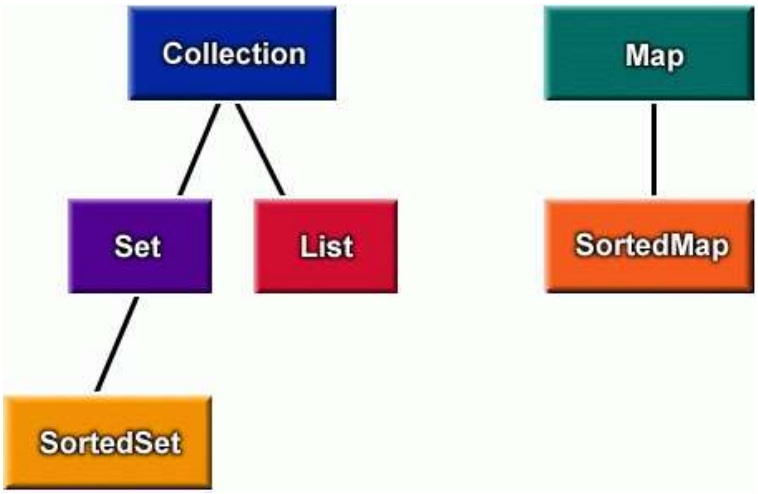
- **Interfacce:** specifiche dei tipi di dati astratti che rappresentano le collezioni.
- **Implementazioni:** realizzazioni concrete delle interfacce delle collezioni.
- **Algoritmi:** classi che realizzano utili algoritmi sulle collezioni, come la ricerca e l'ordinamento.

Con un Collections Framework si hanno i vantaggi tipici delle librerie standard:

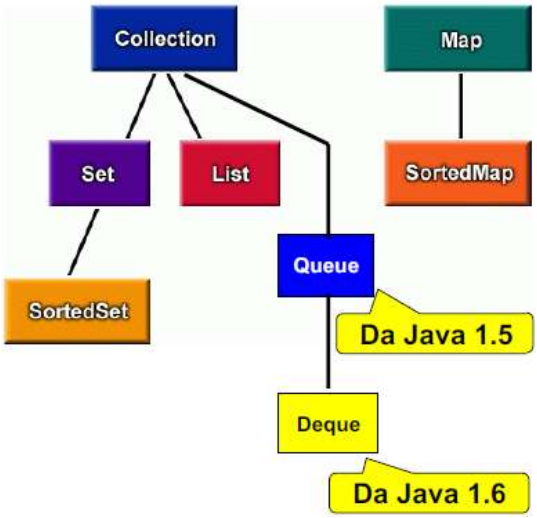
- Si riduce lo sforzo di programmazione;
- Si aumenta l'efficienza e la qualità dei programmi;
- Si favorisce il riuso di programmi.

Java 1.2 fornisce un **framework** completo a supporto delle collezioni con diversi tipi di collezioni e diverse implementazioni per ogni tipo. Le strutture contenitore sono ora suddivise in due categorie: **collections** e **maps**. Questi oggetti permettono di effettuare sia la cosiddetta **programmazione generica** (permettono di scrivere algoritmi generici che funzionano per qualsiasi tipo di oggetto) sia di avere un forte controllo sui tipi istanziando classi ***parametrizzate*** rispetto al tipo di dato. Gli iteratori, poi, permettono di attraversare la collezione attraverso i metodi `next()` e `hasNext()`. Tutte le definizioni sono nel package java.util.

Java Collections Framework: core collection interfaces
(Le principali interfacce della piattaforma JCF)



- **Collection**: gruppo generico di oggetti/elementi; definisce metodi basici indipendenti da molteplicità o ordinamento degli elementi
- **Set**: collezioni senza ripetizione (astrazione matematica corrispondente: *serie*)
- **List**: collezioni i cui elementi sono ordinati (ripetizioni possibili)
- **SortedSet**: insiemi che mantengono ordinati i propri elementi (rispetto all'*ordinamento naturale* determinato da **compareTo**)
- **Map (dizionari):** associazioni chiavi-valori (**HashTable**); le chiavi di inserimento/ricerca sono univoche e si riferiscono ad unico valore inserito
- **SortedMap:** Map con chiavi ordinate



Si nota che mancano alcune altre collezioni notevoli (es: **code**) ma il framework è estensibile e chiunque può fornire un'implementazione personalizzata per un certo tipo di collezione ad esempio una lista implementata da una tabella SQL.

Una descrizione della **struttura di dati dinamica di coda** è contenuta nel package **java.util** ed è rappresentata dall'interfaccia Queue.

L'**interfaccia Queue** non può essere usata per generare degli oggetti, ma impone, alle classi che la utilizzano di implementare i seguenti metodi *astratti*:

Metodo	Descrizione
add (oggetto)	Inserisce un oggetto alla fine della coda.
element()	Restituisce l'oggetto all'inizio della coda senza rimuoverlo.
offer(oggetto)	Inserisce un oggetto alla fine della coda.
peek()	Restituisce l'oggetto all'inizio della coda senza rimuoverlo.
poll()	Restituisce l'oggetto all'inizio della coda e lo rimuove.
remove()	Restituisce l'oggetto all'inizio della coda e lo rimuove.

Un *esempio di implementazione* dell'interfaccia Queue, in cui sono presenti i precedenti metodi, è la **classe LinkedList**. Questa classe, oltre a fornire i metodi di gestione della coda, implementa i metodi usati dalla **struttura di dati di lista concatenata** esemplificati in semplici applicazioni:

Queue<String> codaAttesa = new LinkedList<String>(); // crea e inizializza una **coda**

```
String name = codaAttesa.remove(); // estrae in testa ed elimina recuperando in stringa name
        name = codaAttesa.poll();// estrae in testa ed elimina recuperando in stringa name
                                 // torna null se coda vuota senza lanciare eccezioni
        name = codaAttesa.element();// ritorna elemento in testa senza eliminare
                                 // lancia NoSuchElementException in caso di coda vuota
         name = codaAttesa.peek(); // ritorna elemento in testa senza eliminare
                                 // ritorna null se coda vuota
Boolean flag = codaAttesa.offer("Rajeev");// aggiunge in fondo alla coda se possibile
                                 // (non violando la capacità specificata per la coda)
                                 // restituendo true, altrimenti restituisce false.
        flag = codaAttesa.add("Rajeev"); // aggiunge in fondo alla coda
                                 // e lancia IllegalStateException
                                 // in caso di fallimento
```

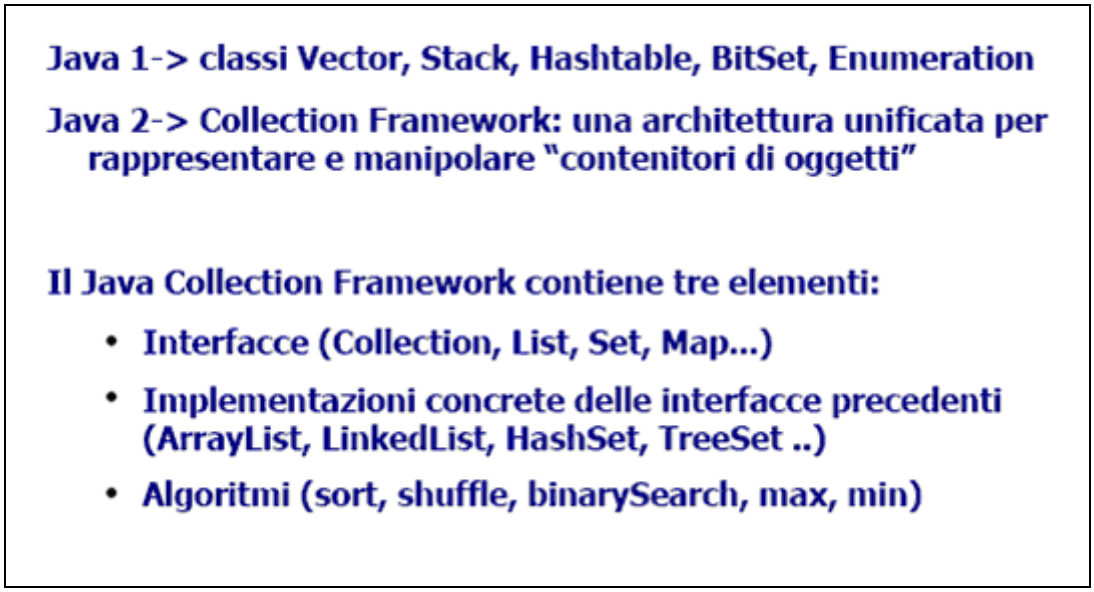
L'elemento rimosso dalla coda dipende dalla *policy di ordinamento* specificata per la coda, che può essere differente da implementazione a implementazione. Esempi di implementazioni di Queue sono: `ArrayBlockingQueue<E>`, `ArrayDeque<E>` e `PriorityQueue<E>`.

Il nome di ogni collezione segue la convenzione <*implementazione*><*tipo*> dando origine alle seguenti classi (si veda il tutorial: http://java.sun.com/docs/books/tutorial/collections/index.html) :

		Implementations				
		Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**	**Hash Table + Linked List**
Interfaces	**Set**	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Ad esempio, un **dizionario** (detto ***Map***) ha tre implementazioni:

- come *hash table*, adatto nei casi generali;
- come array, utile se il dizionario è di ridotte dimensioni e garantisce prestazioni migliori su operazioni di creazione e iterazioni;
- come *albero bilanciato*, che impone un ordinamento agli elementi e offre prestazioni migliori per le operazioni di ricerca ma è più costoso per le inserzioni di elementi.



Le classi Hashtable e Vector del JDK 1.1 (uniche due classi a supporto delle collezioni in tale versione dove era però del tutto assente una differenziazione sull'implementazione) sono ancora disponibili anche se è preferibile ora usare le equivalenti HashMap e ArrayList rispettivamente.

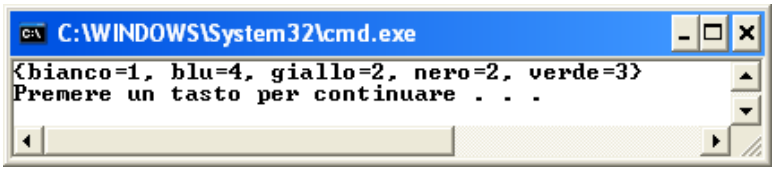
Una caratteristica delle nuove collezioni è di offrire metodi **non sincronizzati**, *più performanti* rispetto a quelli *synchronized* delle classi Hashtable e Vector che privilegiavano la sicurezza nell'impedire ***accessi concorrenti*** a scapito dell'**efficienza** visto che il codice sincronizzato è decisamente penalizzante in Java. Se comunque le collezioni possono avere ***accessi concorrenti*** il framework fornisce delle classi di *avvolgenti* (synchronization wrappers) che si occupano di accedere in modo sincronizzato ai metodi *unsynchronized* di una collezione.

Esempio d'uso di ArrayList (accesso agli elementi della lista mediante indice di posizione)

```
// TestArrayList.java: programma di test per collection ArrayList <E>

import java.util.*;
import java.io.*;

public class TestArrayList {

    public static void main( String args[] ) {
    String[] colori = { "blu", "verde", "blu", "blu", "giallo", "nero" };

    ArrayList <String> lista = new ArrayList < String > ( );

    for (int i=0; i<colori.length; i++)
        lista.add( colori[i] );

    System.out.println( lista.size() );

    ListIterator iter = lista.listIterator();          // interfaccia che eredita da Iterator <E>
                                                       // si recupera la posizione corrente
                                                       // dell'iteratore che è sempre previous() o next()
                                                       // rispetto all'elemento
    while ( iter.hasNext() ) {
        String coloreCorrente = (String) iter.next();
        System.out.print( coloreCorrente + " ");
    }
    System.out.println( );
    } // fine main
} // fine classe
```

Esempio d'uso di TreeMap (*albero bilanciato* → impone un ordinamento agli elementi e offre prestazioni migliori per le operazioni di ricerca ma è più costoso per le inserzioni di elementi)

```
// TestMap.java: programma di test per collection TreeMap <E>

import java.util.*;
import java.io.*;

public class TestMap {

    public static void main( String args[] ) {
    String[] colori = { "blu", "verde", "blu", "verde", "blu", "giallo", "giallo", "nero", "bianco", "nero",
    "blu", "verde" };

    TreeMap <String, Integer> mappa = new TreeMap <String, Integer> ();

    Integer UNO = 1;                          // dalla 1.5 invece di …. Integer UNO = new Integer(1);
    for ( int i=0; i<colori.length; i++ ) {
        String chiave = colori[i];
        Integer frequenza =(Integer) mappa.get( chiave );
        if (frequenza == null)
            frequenza = UNO;
        else
            frequenza = frequenza + 1 ;      //new Integer ( frequenza + 1 );

        mappa.put( chiave, frequenza );
    } // fine for
    System.out.println( mappa );
    } // fine main
} // fine classe
```



Le interfacce

Un **metodo astratto** è un metodo senza corpo, con un "**;**" dopo l'intestazione.

Una **interfaccia** (**interface**) in Java ha una struttura simile a una classe, ma può contenere **soltanto costanti** e **metodi d'istanza astratti** (quindi non può contenere né *costruttori*, né *variabili statiche*, né *variabili di istanza*, né *metodi statici*).

Si dichiarerà che una classe **implementa** (**implements**) una data interfaccia: in questo caso la classe deve fornire una **realizzazione** per tutti i metodi astratti dell'interfaccia, cioè la classe deve fornire metodi con la stessa firma descritta nell'interfaccia (e con il corpo, naturalmente). La realizzazione di un metodo deve anche rispettare la "specifica" del corrispondente metodo astratto.

Le interfacce possono essere utilizzate:

- per definire **Tipi di Dati Astratti**; si pensi agli *insiemi*, definiti come entità matematiche caratterizzate dalle usuali operazioni (*unione*, *appartenenza*, ...), e a diverse possibili implementazioni come *liste con/senza ripetizioni*, *array*, *tabelle hash*, *alberi binari di ricerca*, ...
- come **contratto** tra chi implementa una classe e chi la usa: le due parti possono essere sviluppate e compilate separatamente
- per **evidenziare funzionalità comuni** a più classi, sopperendo alle limitazioni dell'ereditarietà singola
- per scrivere **programmi generici** (applicabili a più classi), evitando di duplicare il codice.

Regole per l'uso di interfacce

Nell'uso delle interfacce in un programma, ricordarsi delle seguenti regole:

- Possiamo dichiarare una variabile indicando come tipo un'interfaccia:
  ```
  TipoInterfaccia nomeVariabile;
  ```
- Non possiamo istanziare un'interfaccia:
  ```
  TipoInterfaccia nomeVariabile  = new TipoInterfaccia(); // VIETATO
  ```
- Ad una variabile di tipo interfaccia possiamo assegnare solo istanze di classi che implementano l'interfaccia
- Su di una variabile di tipo interfaccia possiamo invocare solo metodi dichiarati nell'interfaccia (o nelle sue "super-interfacce").

Si noti che:

- Le variabili devono essere inizializzate e non possono essere modificate successivamente: anche se non sono dichiarate `final` di fatto sono delle **costanti**;
- I metodi sono tutti **astratti**: infatti al posto del corpo c'è solo un punto e virgola;
- I metodi dichiarati in una interfaccia sono sempre `public`. Di conseguenza, i corrispondenti metodi di una classe che implementa l'interfaccia devono essere `public`.
- Una interfaccia può estendere una o più **interfacce** (non **classi**), indicate dopo la parola chiave `extends`. Per le interfacce non vale la restrizione di "ereditarietà singola" che vale per le classi.

Interfacce come tipi di dati astratti

In Java ogni classe definisce un **tipo di dati**, i cui **elementi** sono le istanze della classe (la cui struttura è determinata dalle variabili d'istanza), e le cui **operazioni** sono i metodi.

Una interfaccia definisce invece un **tipo di dati astratto**, di cui fornisce la **specifica delle operazioni**: la struttura dei suoi elementi e il modo in cui le operazioni sono effettivamente definite verrà determinato dal tipo di dati (la classe) che realizza (**implements**) l'interfaccia.

Generics

I generics sono stati introdotti in Java con il duplice obiettivo:

- permettere la creazione di oggetti e/o metodi con **un controllo esplicito sui parametri** (type-safe objects)
- permettere la creazione di oggetti e/o metodi altamente **parametrizzati**

I generics sono realizzati con un meccanismo di espansione/sostituzione detto Erasure and Translation

Dopo gli opportuni controlli di consistenza tra tipi, il compilatore **cancella** (Erasure) tutte le informazioni relative ai tipi parametrici, e **sostituisce** (Translation) le relative occorrenze con il tipo di oggetto specifico

Nell'uso delle **collezioni** il programmatore deve gestire situazioni di **ambiguità** e deve sapere con certezza il tipo specifico di oggetto che sta manipolando.

```
List<String>stringhe = new ArrayList<String>();

stringhe.add("Pippo");
stringhe.add("Pluto");

String x = stringhe.iterator().next();
```

Osservando nelle API la definizione della interfaccia List:

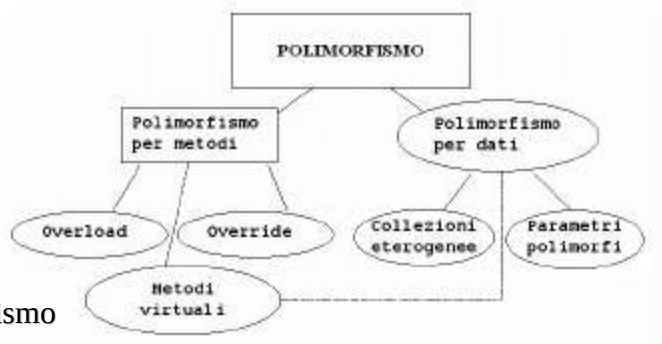
```
public interface List<E> extends Collection<E> {
    .....
    boolean add(E o)      // aggiunge elemento in coda
    E get(int index)      // legge elemento in posizione specificata
}
```

Prima dell'introduzione dei Generics, si usavano contenitori di *Object* (tutti gli oggetti estendono la classe Object) quindi qualsiasi cosa poteva essere inserita in tale struttura (anche tipi primitivi di base tramite le classi wrapper) ma tale flessibilità era a scapito della possibilità di un controllo sui tipi da parte del compilatore. La gestione del corretto uso dell'oggetto e dell'esplicita conversione dello stesso in un Object type-safe era a carico del programmatore attraverso l'uso di un'operazione di **casting**.

Al posto di Object abbiamo un "oggetto" E, che in realtà è nient'altro che un **placeholder**.

L'oggetto viene tipizzato in fase di definizione della collection: il compilatore sostituisce ognuno dei placeholder con l'oggetto specifico che caratterizza la collection .

Le collezioni eterogenee sono un tipo di polimorfismo



"il polimorfismo in Java"

Creazione di classi che usano Generics

```
class Mem<E> {
     private ArrayList<E> e = new ArrayList<E>();
     public void push(E item) {
           e.add(item);
     }
     //ecc....
     }
```

- si possono passare più tipi nelle parentesi angolari:

```
class TabellaValue<K, V> {
                                    // K e V possono essere usati ovunque,
                                    // come tipo di dato, di ritorno di parametro ecc...
 }
```

- anche le interfacce possono usufruirne:

```
class CodaReale<E> implements Coda<E> {
                                                    //ecc...
}
```

Creazione di un oggetto generic

```
import java.util.*;

public class VectorType<T> {

   private Vector<T>vettore = new Vector<T>();

   public void add(T elemento) {
        vettore.add(elemento);
   }
   public void printAll() throws Exception {
            Enumeration<T>en = vettore.elements();  // possibile uso di Enumeration che deve essere generic
            while(en.hasMoreElements()) {
               System.out.println(en.nextElement());
             }                                      // … codice da migliorare con uso di iteratore
   }
   public void printAllNewLoop() throws Exception {
                                                    // per scandire gli elementi
          for(T elemento:vettore) {                 // nuova sintassi for-each
            System.out.println(elemento);
          }
   }
}
                                                    // … codice migliorato con uso di iteratore

   public void printAll()throws Exception {

         Iterator iter = vettore.iterator();
         while(iter.hasNext()) {
               System.out.println(iter.next());
         }          // durante l'iterazione un iterator può anche rimuovere elementi, un Enumeration no
   }
```

Uso di una mappa K,V

```
                                                    // Libro classe definita da utente
Map<Integer,Libro>mappa = new HashMap<Integer,Libro>();

mappa.put(1,new Libro("Satyricon"));
mappa.put(2,new Libro("Iliade"));

Libro libro = mappa.get(1);
```

Esempio con implementazione del modello *hash table*, adatto nei casi generali (senza ordinamento)

```
import java.util.*;

public class EnvMap {
    public static void main (String[] args) {
        HashMap<String, String> env =
                    new HashMap<String, String>(System.getenv());
                                                    // recupero le variabili di sistema
         for (String envName : env.keySet()) {  // nuova sintassi for-each
            System.out.printf ("%s=%s%n", envName,
                                    env.get(envName));
                                                // stampo con formattazione le variabili di sistema
        }                                       //  del processo che esegue il metodo System.getenv()

    }
}
```

nb: il metodo **public static Map<String, String> java.lang.System.getenv()** restituisce un oggetto di tipo java.util.**Map** contenente le variabili di sistema correnti e i loro valori. E' possibile quindi un assegnamento diretto senza costruire un'istanza di classe (HashMap o TreeMap) che implementa l'*interfaccia **Map***:

```
import java.util.*;
public class StampaVariabili {

    public static void main(String args[]){

    Map <String, String> env = System.getenv();  // assegnamento diretto

    for (String envName : env.keySet())  // nuova sintassi for-each
                System.out.printf ("%s=%s%n", envName, env.get(envName));
    }
}
```

http://java.sun.com/docs/books/tutorial/collections/interfaces/map.html

Generics *wildcards*

Nella sintassi dei generics viene usata la keyword **?** come c**arattere Jolly (**wildcard).

Se infatti volessimo definire un metodo che accetti collezioni di un qualsiasi oggetto non possiamo usare Collection <Object> coll, in quanto Object non è più supertype di tutti gli oggetti.

La sintassi corretta è con l'uso del wildcard ?

```
Collection <?> coll
```

(*collezione di oggetti sconosciuti la cui definizione è al momento della creazione*)

Altri usi di wildcard (bounded wildcards):

<? extends type> indica tutti i tipi che ereditano da type.

<? super type> tutte le superclassi di type

Uso di Generics *wildcards*

```
import java.util.*;
public class GenericsUtility {
 public static void printAllCollections(Collection<?>collezione)
       throws Exception {
       for(Object oggetto:collezione) {
            System.out.println(oggetto);
    }
 }
 public static void printAllExtendsLibro(List<? extends
    Libro>libri)
    throws Exception {
    for(Object oggetto:libri) {
         System.out.println(oggetto.toString());
    }
 }
 public static void merge2Collection(  Collection<? extends
    Libro>coll1,Collection<? extends Libro>coll2)  throws Exception
    {
    ArrayList<Libro>dest = new ArrayList<Libro>();  // con uso classe Libro
    dest.addAll(coll1);
    dest.addAll(coll2);
    for(Object oggetto:dest) {
          System.out.println(oggetto.toString());
    }
 }
}
```

Glossario

Tipo di dato astratto (ADT)

• Astrazione sui dati che non li classifica in base alla rappresentazione (*implementazione*), ma in base alle caratteristiche (informazioni contenute) e al comportamento atteso (*specifica*).

• Il comportamento dei dati è espresso in termini di un insieme di operazioni applicabili a quei dati (*interfaccia*)

• Le operazioni dell'interfaccia sono le sole che si possono utilizzare per creare, modificare e accedere agli oggetti

- L'implementazione della struttura dati non può essere utilizzata al di fuori della definizione della struttura.

• ADT incapsula tutte le operazioni che manipolano i valori del tipo:

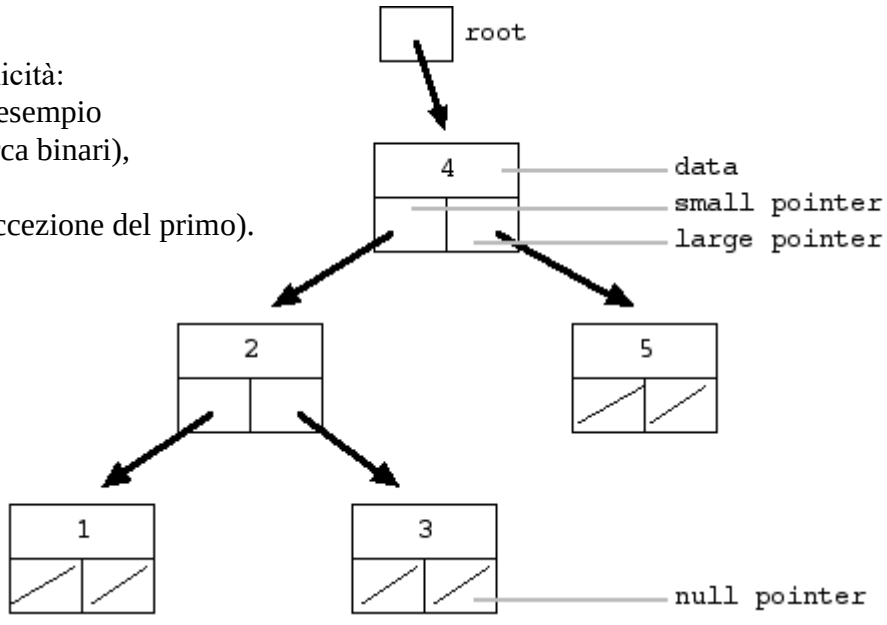
- esporta
  - il nome del tipo
  - l'interfaccia: tutte e sole le operazioni per manipolare oggetti (cioè dati) del tipo e la loro specifica
- nasconde
  - la struttura del tipo
  - l'implementazione delle operazioni

ADT lineari

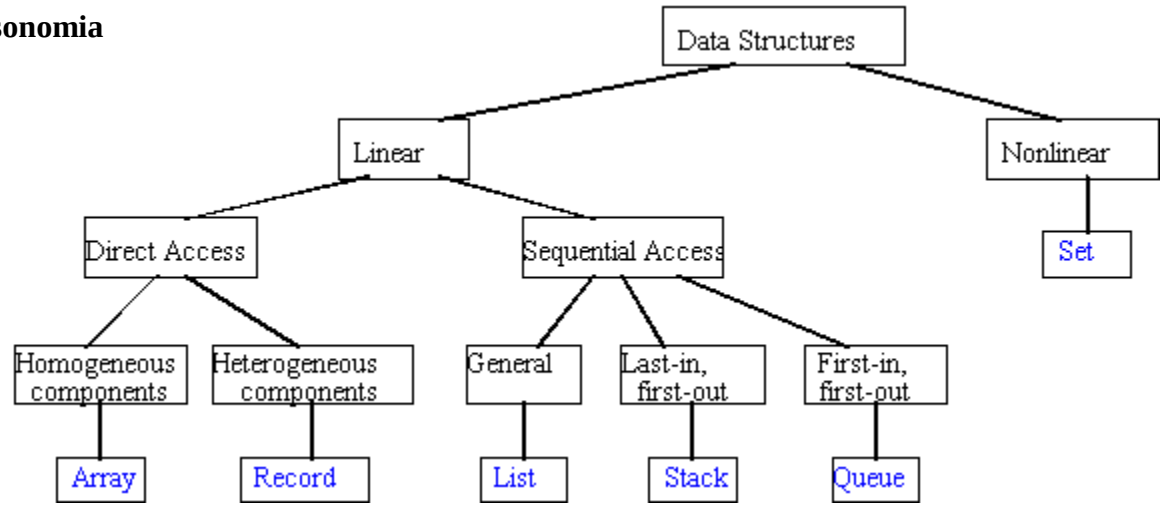
• Ogni elemento della struttura ha un **unico successore** ed un **unico predecessore** (ad eccezione del primo). Esempi: Pila e Coda

ADT non lineari

• Senza lo stretto vincolo sull'unicità:
un elemento della struttura, ad esempio
nel caso di ABR (alberi di ricerca binari),
può avere **più successori**
ed un unico predecessore (ad eccezione del primo).

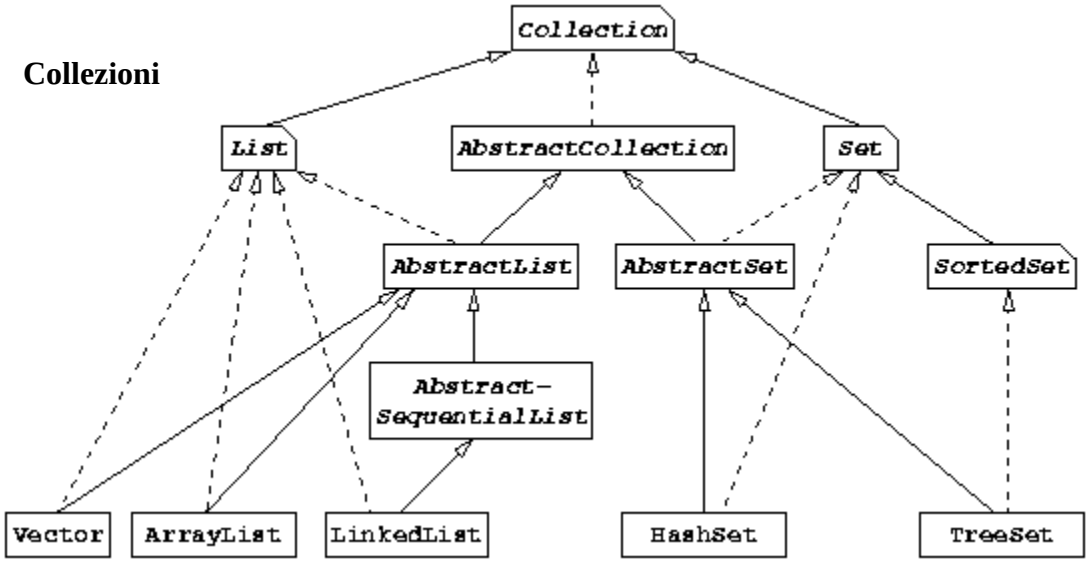


Tassonomia

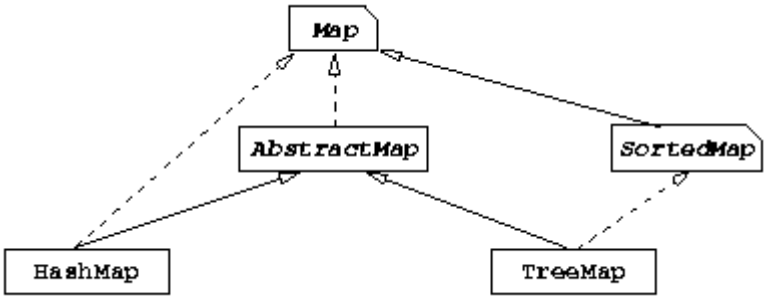


Strutture dati nella piattaforma
Java **C**ollections **F**ramework
(JCF):

Collezioni



Map



Slide con esempi: **Collections** Java (*vantaggi nell'uso di Generics*)