UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA
DIPARTIMENTO DI ECCELLENZA

Università di degli Studi di Salerno
Dipartimento di Informatica

Programmazione ad Oggetti

a.a. 2023-2024

Classi Contenitore

Docente: Prof. Massimo Ficco

E-mail: *mficco@unisa.it*

Classi Contenitore

- In Java è possibile definire array, ovvero sequenze di elemento dello stesso tipo e con lunghezza fissata all'atto dell'istanziazione di una variabile **array**, e non più modificabile durante il ciclo di vita della variabile. Tale costrutto non riesce a soddisfare tutte le esigenze dei programmatori, pertanto Java dispone di apposite classi per astrarre casi in cui il tipo array non è adeguato.
- Un esempio pratico è il caso di una sequenza di caratteri che prende il nome di **stringa**. Java dispone di un'apposita classe per la gestione di stringhe, chiamata String. Le istanze di String sono stringhe non modificabili e sono creabili con assegnazione di una sequenza di caratteri tra doppi apici ("") o con i costruttori della classe.
  - public String () - crea una stringa vuota;
  - public String (char[] value) - crea una stringa che contiene i caratteri di value;

StringBuffer

- Le stringhe implementate come istanze di String sono oggetti non modificabili né estendibili. Questo vuol dire che ogni qualvolta viene assegnato un nuovo valore ad una stringa (in un'operazione di concatenazione, rimozione o aggiunta caratteri) in realtà vengono create nuove stringhe. In Java è possibile realizzare stringhe modificabili ed estendibili con istanze della classe **StringBuffer**, creabili per mezzo dei costruttori della classe:
  - **public StringBuffer ()** - crea uno StringBuffer vuoto, con lunghezza iniziale 16 (default);
  - **public StringBuffer (int lenght)** – come prima ma con lunghezza iniziale length;
  - **public StringBuffer (String str)** - contenuto iniziale: str. Consente di usare indirettamente i costruttori della classe String.
- Alcuni metodi:
  - **public String toString()** - restituisce la stringa corrispondente;
  - **public StringBuffer append (Tipo value)** - concatena la rappresentazione testuale dell'argomento al proprio contenuto.

StringBuffer

- La classe StringBuffer non è molto usata in modo esplicito dai programmatori, ma lo è dal compilatore Java. In particolare, i metodi append sono utilizzati dal compilatore per implementare l'operatore di concatenazione di stringhe '+'.

- Esempio: consideriamo il comando

  x = "a" + 4 + 'c';

- viene compilato come se fosse l'espressione:

  x = new StringBuffer().append("a").append(4).append('c').toString();

- che crea un oggetto di classe StringBuffer vuoto, quindi ci concatena in sequenza "a", 4 e 'c', e infine restituisce la stringa corrispondente.

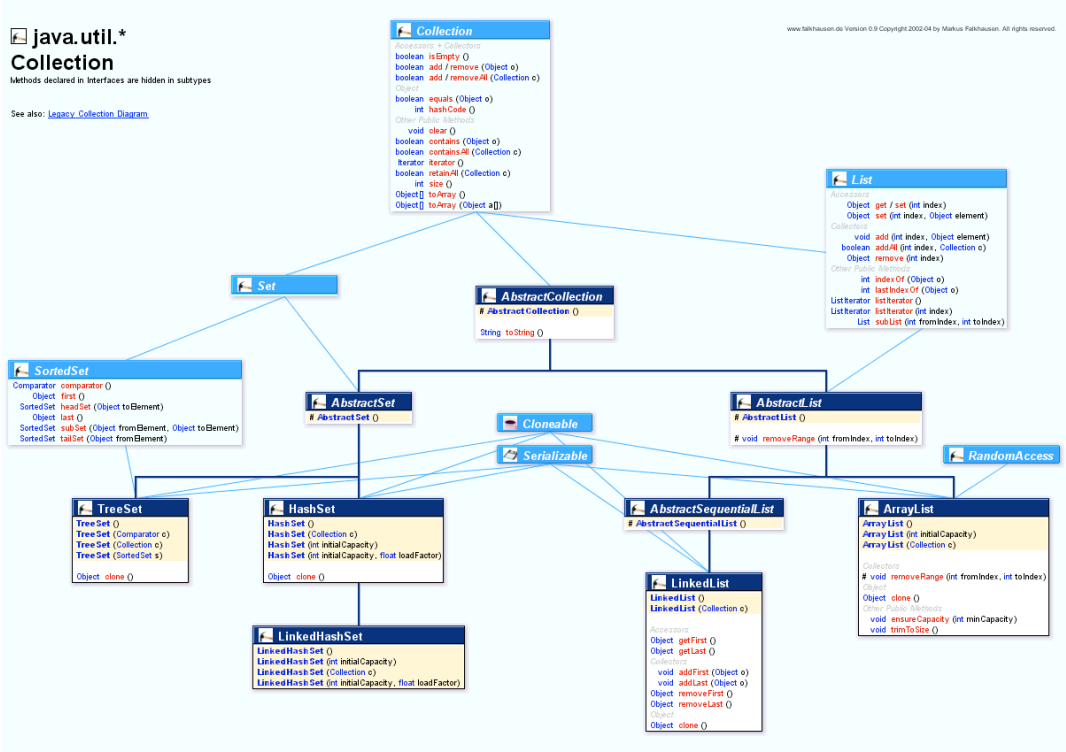
Classi Contenitore

- La libreria **java.util** offe un insieme completo di classi **contenitore**, i cui tipi base sono **List**, **Set**, **Queue** e **Map**. A differenza degli array, le classi contenitore in Java si ridimensionano automaticamente, e permettono di inserire un numero di oggetti arbitrario senza definire la dimensione all'interno del programma.

- Un contenitore (chiamato anche **container** o **collezione**) è un oggetto che raggruppa elementi multipli in una singola unità ed è utilizzato per memorizzare, recuperare e manipolare dati, per trasmetterli da un metodo ad un altro. Le varie classi contenitore sono state introdotte a partire dalla release 1.2, nel contesto del cosiddetto collection framework.
- Tale framework per i contenitori è costituito da
  - Interfacce, ovvero i tipi di dato astratti che rappresentano le classi contenitori, permettono di manipolare i contenitori indipendentemente dettagli della rappresentazione e sono organizzate a formare una gerar

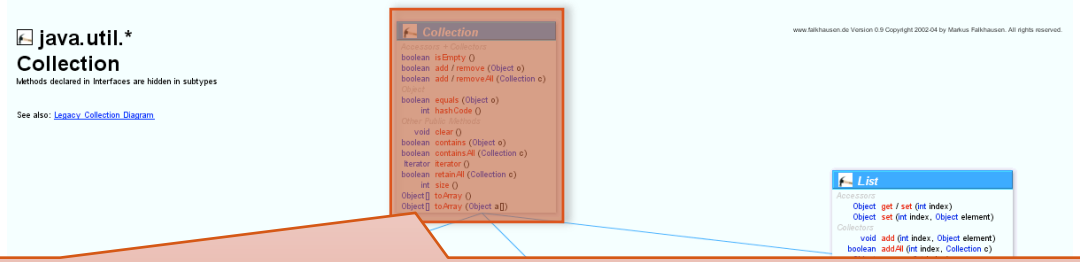
Classi Contenitore

- La libreria scinde la “gestione degli oggetti” in due concetti:
  - **Collection** - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List e Set);



Classi Contenitore - Collection

- La libreria scinde la “gestione degli oggetti” in due concetti:
  - Collection - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List e Set);



Collection è l’interfaccia root di una gerarchia di classi per le sequenze di elementi e rappresenta il minimo comun denominatore che tutte le collezioni implementano. Alcune implementazioni ammettono duplicati altre no, oppure alcune offrono l'ordinamento degli elementi altre no. JDK non ha implementazioni di tale interfaccia, ma vengono fornite implementazioni delle sue sotto-interfacce come Set e List.

Classi Contenitore - Set

- La libreria scinde la “gestione degli oggetti” in due concetti:
  - Collection - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List e Set);

Set rappresenta la collezione che non può contenere duplicati ed è l'astrazione dell’insieme matematico.

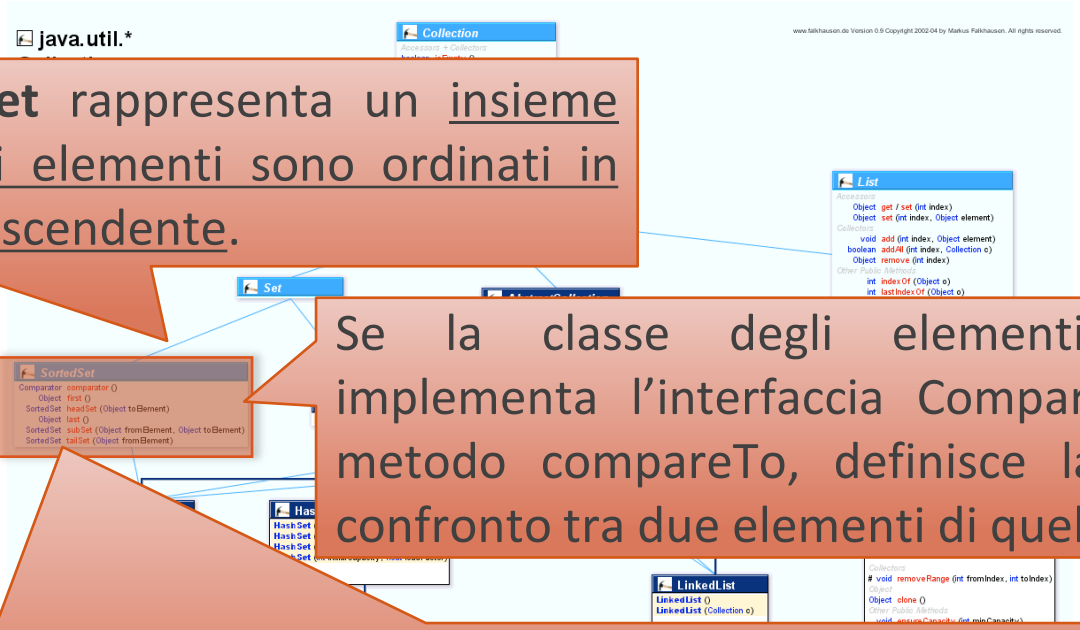
Classi Contenitore - List

- La libreria scinde la “gestione degli oggetti” in due concetti:
  - Collection - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List e Set);

List rappresenta la collezione ordinata (detta anche sequenza) e può contenere elementi duplicati. Si accede agli elementi mediante un indice intero (detta posizione).

Classi Contenitore - SortedSet

- La libreria scinde la “gestione degli oggetti” in due concetti:
  - Collection - una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole (List e Set);



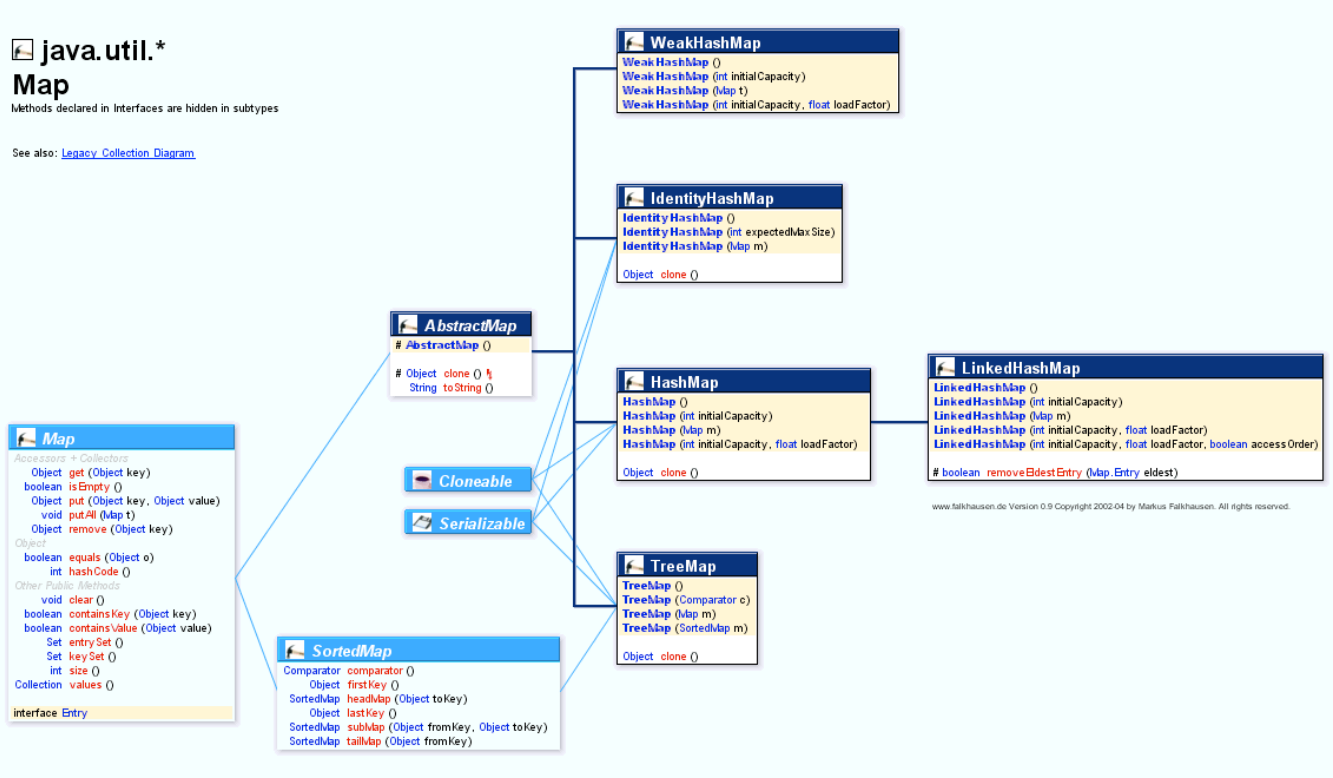
SortedSet rappresenta un insieme dove gli elementi sono ordinati in ordine ascendente.

Se la classe degli elementi nell’insieme implementa l’interfaccia Comparable, allora il metodo compareTo, definisce la relazione di confronto tra due elementi di quella classe.

Oppure bisogna esplicitare la relazione di confronto fornendo all’insieme un oggetto che implementa l’interfaccia Comparator.

Classi Contenitori - Map

- La libreria scinde la “gestione degli oggetti” in due concetti:
  - **Map** - un gruppo di coppie chiave-valore indicanti oggetti, per permettono di recuperare un valore mediante la chiave ad esso associata.



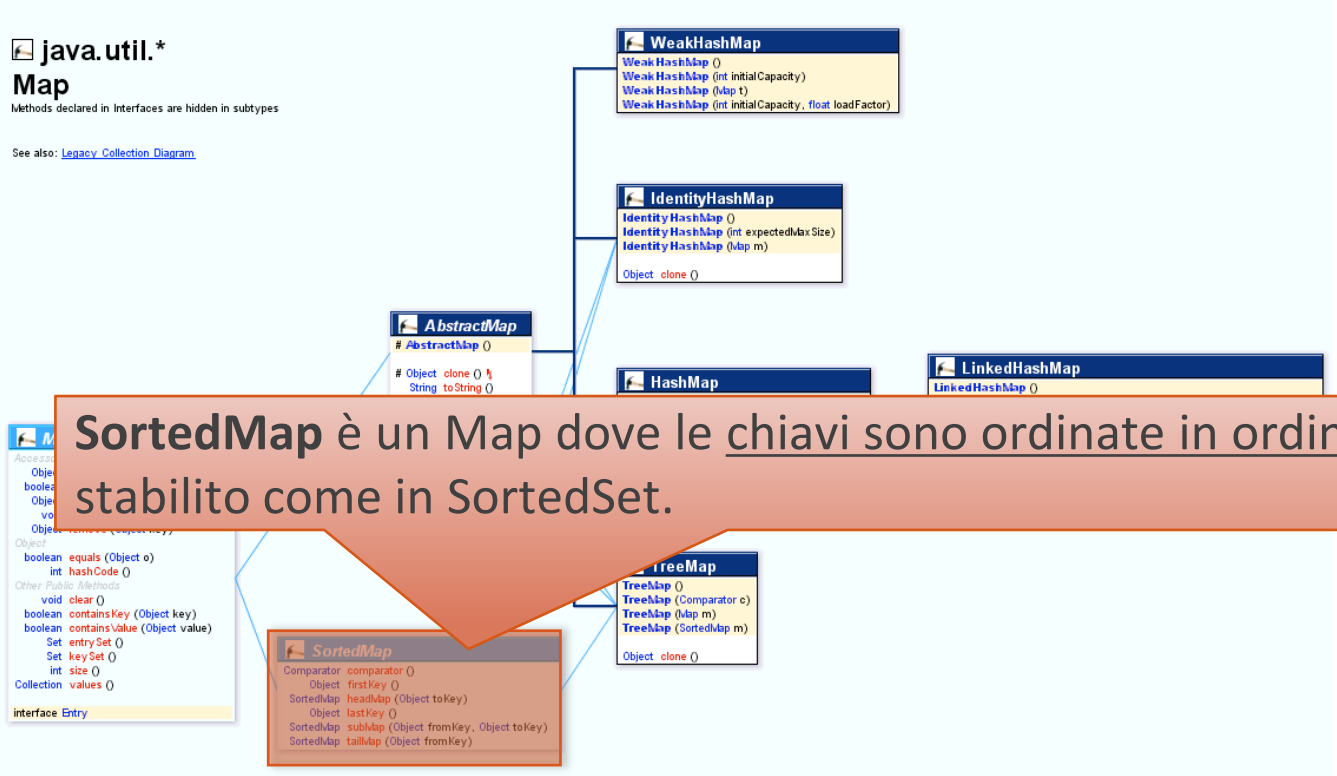
Classi Contenitori - Map

- La libreria scinde la “gestione degli oggetti” in due concetti:
  - Map - un gruppo di coppie chiave-valore indicanti oggetti, per permettono di recuperare un valore mediante la chiave ad esso associata.

Map è l'interfaccia radice delle classi che rappresentano una raccolta di oggetti che mappano una chiave ad un valore, e non possono contenere chiavi duplicate, ovvero una chiave mappa un solo valore.

Classi Contenitori - Map

- La libreria scinde la “gestione degli oggetti” in due concetti:
  - Map - un gruppo di coppie chiave-valore indicanti oggetti, per permettono di recuperare un valore mediante la chiave ad esso associata.



SortedMap è un Map dove le chiavi sono ordinate in ordine ascendente, stabilito come in SortedSet.

Classi Contenitori

- Di seguito sono riportate le principali implementazioni di Set, List e Map:

Interfacce	Implementazioni			
	Hash Table	Resizable Array	Balanced Tree	Linked List
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

- Sono presenti almeno due implementazioni per ogni interfaccia. Le implementazioni primarie sono **HashSet**, **ArrayList** e **HashMap**.
- TreeSet e TreeMap implementano rispettivamente SortedSet e SortedMap.
- In aggiunta, sono presenti due classi, **Vector** e **Hashtable** precedenti all'introduzione del framework, che sono state mantenute e modificate per implementare le interfacce del framework.

List

- List realizza una collezione di elementi in cui possono esserci elementi duplicati, e rispetto a Collection ci sono delle operazioni aggiuntive:
  - Accesso posizionale (si parte da 0) ovvero manipolazione degli elementi in base alla posizione nella lista:

    Object get(int index);
  - Ricerca di un determinato oggetto e ritorno della posizione numerica:

    int indexOf(Object o);
  - Estrazione di sotto-liste:

    List subList(int fromIndex, int toIndex);
- Le possibili implementazioni di questa interfaccia sono ArrayList, LinkedList e Vector. LinkedList dispone di metodi di inserimento ed estrazione impiegabili per la realizzazione di code e pile. Esistono l'interfaccia Deque per la realizzazione di pile e code con classi ad hoc.
- Due Liste sono uguali se gli elementi sono gli stessi nello stesso ordine.

Classi Contenitori

- **ArrayList** e **Vector** sono due classi simili per la realizzazione di array estendibile, con un analogo API, ma sono caratterizzate da differenze:
  - Internamente, ArrayList e Vector gestiscono gli elementi con un array. Se quando si istanzia la lista non si indica la grandezza, ArrayList e Vector cambiano dinamicamente la grandezza del loro array interno. La differenza sostanziale tra ArrayList e Vector, in questo caso, sta nella nuova grandezza che l'array interno assume: ArrayList lo aumenta del 50%, Vector lo raddoppia.
- Ecco perché si sconsiglia di istanziare una lista e incominciare ad aggiungere gli elementi, ma è preferibile settare una grandezza massima, tale da evitare di pagare il resize dinamico dell'array interno.
- Vector ha però un vantaggio: se si conosce il rate con cui crescono gli inserimenti è possibile settare il valore di incremento dell'array interno (con le API a tempo di costruttore).

Classi Contenitori

- Sia **ArrayList** che **Vector** permettono di recuperare un elemento in una determinata posizione e di aggiungere o rimuovere in coda al costo di O(1).
- Aggiungere o rimuovere elementi in una qualsiasi altra posizione ha costo lineare O(n), in quanto è necessario lo shift degli elementi successivi.
- In particolare, se risulta quest'ultima l'operazione dominante, allora sarebbe utile scartare sia ArrayList che Vector ed utilizzare **LinkedList**, che aggiunge o rimuove elementi al costo di O(1), in cambio di un peggiore costo per l'indicizzazione e di un maggiore garbage (viene costruito un oggetto interno per ogni elemento).

Collection

- Metodi offerti:
  - **int size()** – restituisce la dimensione della collezione;
  - **boolean isEmpty()** – restituisce true se la collezione non ha elementi, false in caso contrario;
  - **boolean contains(Object element)** – verifica che element è incluso nella collezione;
  - **boolean add(Object element)** – aggiunge element alla collezione;
  - **boolean remove(Object element)** – rimuove element dalla collezione;
  - **boolean containsAll(Collection c)** – verifica se c è inclusa nella collezione
  - **boolean addAll(Collection c)** – aggiunge tutti gli elementi di c nella collezione;
  - **boolean removeAll(Collection c)** - rimuove tutti gli elementi di c dalla collezione;
  - **boolean retainAll(Collection c)** – mantiene nella collezione solo gli elementi che sono anche in c;
  - **void clear ()** – elimina tutti gli elementi dalla collezione;
  - **Object[] toArray()** - restituisce gli elementi della collezione come un array;

Collection

- **Iterator iterator()** – fornisce un iteratore alla collezione, definito per mezzo della seguente interfaccia:

```
interface Iterator  {
   boolean hasNext();      // Verificare se esiste un prossimo elemento
   Object next();          // Restituisce l'elemento corrente
   void remove();          // Elimina l'elemento corrente (opzionale)
}
```

- Tale interfaccia è specializzata per la collection specifica di interesse:

```
interface ListIterator extends Iterator { … }
```

Collection

- Per convenzione tutte le implementazioni di Collection presentano un costruttore con argomento un'istanza di un'implementazione di Collection, per l'inizializzazione di una collezione a partire da un'altra:

  List l = new ArrayList(c); // Con c oggetto di una qualunque collezione

- Le classi contenitori in Java contengono oggetti di tipo Object e sue sotto-classi:
  - **boolean contains(Object element);**
  - **boolean add(Object element);**
  - **boolean remove(Object element);**

- E' necessario effettuare un casting quando si recupera l'oggetto dalla collezione, ed è possibile contenere tipi eterogenei.

Collection

```
public class CollectionsTest {
   public static void main( String[] args ) {
      Collection c = new ArrayList();  // ArrayList c = new ArrayList(); List c = new ArrayList();
      c.add( "ten" );
      c.add( "eleven" );
      System.out.println( c );
      Object[] array  = c.toArray();
      for (int i = 0; i < array.length; i++ ) {
            String element = ( String ) array[ i ];
            System.out.println( "Elemento di array:" + element );
            }
      } }
```

Collection

```
public class CollectionsTest {
   public static void main( String[] args ) {
      Collection c = new ArrayList();
      c.add( "ten" );
      c.add( "eleven" );
      System.out.println( c );
      Object[] array  = c.toArray();
      for (int i = 0; i < array.length; i++ ) {
            String element = ( String ) ar
```

```
nt ); }
```

Proviamo una soluzione alternativa per accedere agli elementi di una collezione.

Collection

```
public class CollectionsTest {
   public static void main( String[] args ) {
      Collection c = new ArrayList();
      c.add( "ten" );
      c.add( "eleven" );
      System.out.println( c );
      Object[] array  = c.toArray();
      for ( Iterator i = c.iterator(); i.hasNext(); ) {
            String element = ( String ) i.next();
            System.out.println( "Elemento di array:" + element );  }
      } }
```

Collection

```
public class Dog  {
  private int dogNumber;
  public Dog(int i) { dogNumber = i; }
  public void id() { System.out.println("Dog #" + dogNumber);   } }

public class Cat {
  private int catNumber;
  public Cat(int i) { catNumber = i; }
  public void id() { System.out.println("Cat #" + catNumber );   } }

public class CatsAndDogs {
  public static void main(String[] args) {
     List cats = new ArrayList();
     for(int  i = 0; i < 7; i++)
        cats.add(new Cat(i));      //Nessun problema mettere un cane
     cats.add(new Dog(7));
     for(int i = 0; i < cats.size(); i++)
        // Si rileva il cane solo a tempo di esecuzione
        ((Cat) cats.get(i)).id();   } }
```

Collection

```
public class Dog  {
  private int dogNumber;
  public Dog(int i) { dogNumber = i; }
  public void id() { System.out.println("Dog #" + dogNumber);   } }
public class Cat {
  private int
  public Cat(
  public voi
public class
  public stat
    List cats
    for(int  i = 0; i < 7; i++)
      cats.add(new Cat(i));    //Nessun problema mettere un cane
    cats.add(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
       // Si rileva il cane solo a tempo di esecuzione
       ((Cat) cats.get(i)).id();   } }
```

Si perde l'informazione sul tipo dell'oggetto che viene inserito in un contenitore.
Un contenitore colleziona riferimenti a Object che è la radice di tutte le classi e può in tal modo collezionare oggetti di qualsiasi tipo.

Collection

```
public class Dog  {
  private int dogNumber;
  public Dog(int i) { dogNumber = i; }
  public void id() { System.out.println("Dog #" + dogNumber);   } }
public class Cat {
  private int catNumber;
  public Cat(int i) { catNumber = i; }
  public voi
public class
  public sta
    List cats
    for(int i
      cats.a
    cats.add
    for(int i
      // Si rileva il cane s      mpo di esecuzione
      ((Cat) cats.get(i)).id();  } }
```

Poiché le informazioni sul tipo vengono perse, la sola cosa che il contenitore sa è che contiene un riferimento a un oggetto. È quindi necessario eseguire un cast al tipo corretto prima di utilizzarlo, altrimenti si genera un eccezione.

*Collection

- Java 5 risolve tali problemi utilizzando i generics, con le collezioni parametrizzate sul tipo dei dati da contenere:

```
List<Integer> l = new ArrayList<Integer>();
```

Set

- Set dispone degli stessi metodi dell'interfaccia Collection, ma sono proibiti elementi duplicati e nulli nella sequenza.

- Possibili implementazioni sono **HashSet**, **TreeSet** e **LinkedHashSet** dove si usa:
  - HashSet per i Set in cui la velocità di consultazione è importante;
  - TreeSet per mantenere un Set ordinato sostenuto da una struttura ad albero;
  - LinkedHashSet possiede la stessa velocità di consultazione di un HashSet, ma mantiene inoltre l'ordine nel quale gli elementi sono stati inseriti utilizzando internamente una lista concatenata.

Set

```
import java.util.*;
public class FindDups  {
      public static void main(String args[]) {
            Set s = new HashSet();
            for (int i=0; i<args.length; i++)  {
                  if (!s.add(args[i]))
                        System.out.println("Duplicate detected: "+args[i]);
            }
            System.out.println(s.size()+" distinct words detected: "+s);
      }
}

C:> java FindDups i came i saw i left
OUTPUT
Duplicate detected: i
Duplicate detected: i
4 distinct words detected: [came, left, saw, i]
```

Set

```
import java.util.*;
public class FindDups  {
      public static void main(String args[]) {
            Set s = new HashSet();
            for (int i=0; i<args.length; i++)  {
                  if (!s.add(args[i]))
                        System.out.println("Duplicate detected: "+args[i]);
            }
      }
}
```

NOTA: Modificando HashSet in TreeSet si ottiene l'ordinamento

```
C:> java FindDups i came i saw i left
OUTPUT
Duplicate detected: i
Duplicate detected: i
4 distinct words detected: [came, left, saw, i]
```

Set

```
import java.util.*;
public class FindDups2 {
      public static void main(String args[]) {
            Set uniques = new HashSet();  Set dups  = new HashSet();
            for (int i=0; i<args.length; i++)  {
                  if (!uniques.add(args[i]))
                        dups.add(args[i]);  // Aggiungo solo i duplicati
                  uniques.removeAll(dups);
            }
            System.out.println("Unique words: " + uniques);
            System.out.println("Duplicate words: " + dups);
      }
}

C:> java FindDups2 i came i saw i left
OUTPUT
Unique words: [came, left, saw]
Duplicate words: [i]
```

Map

- Non può contenere duplicati delle chiavi, e presenta tre diverse implementazioni: **HashMap**, **TreeMap**, e **Hashtable**. HashMap è da preferire rispetto ad Hashtable perché fornisce prestazioni costanti per l'inserimento e la ricerca di qualunque coppia. TreeMap è basata su una struttura ad albero e mantiene ordinati i dati.
- Come le collezioni, anche le mappe hanno un costruttore con argomento una mappa per l'inizializzazione di una nuova mappa con gli elementi di quella specificata in ingresso.

Classi Contenitori

- Le classi **HashMap** e **HashTable** sono simili, infatti entrambe sono delle tabelle chiave-valore, ma HashTable è sincronizzata ovvero thread-safe, mentre l'HashMap no. In più l'HashMap permette valori null sia per le chiavi che per i valori. Apparentemente quindi sembra che HashTable debba essere più lento di HashMap a causa della sincronizzazione. Infatti, facendo dei test di carico, HashTable è migliore per carichi medio-bassi, mentre HashMap risulta migliore per carichi più alti.

Map

- Metodi offerti:
  - **Object put(Object key, Object value)** – per inserire una coppia chiave valore nella mappa ;
  - **Object get(Object key)** – per ottenere un valore mappato dalla chiave key;
  - **Object remove(Object key)** – per rimuovere a coppia identificata dalla chiave key;
  - **boolean containsKey(Object key)** – per verificare se una chiave è presente nella mappa;
  - **boolean containsValue(Object value)** – per verificare se un valore è presente nella mappa;
  - **int size()** – per ritornare la dimensione della mappa;
  - **boolean isEmpty()** – per verificare se la mappa ha elementi;
  - **void putAll(Map t)** – per inserire in una mappa tutti gli elementi di un'altra mappa;
  - **void clear()** – per eliminare tutti gli elementi;
  - **Set keySet()** – per ottenere l'insieme di tutte le chiavi nella mappa;
  - **Collection values()** – per ottenere la lista dei valori nella mappa;

HashMap<k,v>

HashMap è un'implementazione dell'interfaccia Map che fornisce una struttura dati per archiviare i dati in coppie valore-chiave

Non esiste un ordinamento degli elementi

```
Map<KeyType, ValueType> myMap = new HashMap<KeyType, ValueType>();
```

```
Map<String,Integer> myMap = new HashMap<String,Integer>();
```

HashMap<k,v>

- Inserire dei valori

```
myMap.put("key1", 1);
myMap.put("key2", 2);
```

- Ottenere dei valori

```
myMap.get("key1"); //return 1 (class Integer)
```

- Controllare se la chiave è nella mappa o no.

```
myMap.containsKey(varKey);
```

- Controlla se il valore è nella mappa o no.

```
myMap.containsValue(varValue);
```

TreeMap<k,v>

E’ utilizzata in situazioni in cui è richiesto un ordinamento sulle chiavi

Esempio, realizziamo una TreeMap che memorizzi le coppie cognome-nome di una persona ordinandole alfabeticamente sul cognome

Si tratta di una classe la cui complessità di tempo per le varie operazioni è logaritmica nella dimensione della mappa.

Map

```
public class TestMap  {
    public static void main( String[] args ) {
        Map map  = new HashMap();
        map.put( "key1", "value1" );
        map.put( "key2", "value2" );
        map.put( "key3", "value1" );
        Object value = map.get( "key1" );
        System.out.println( "Valore: " + value );
        System.out.println( "Valore: " + map.get( "key2" ) );
    }
}
```

Map

```
public class Freq {
    private static final Integer ONE = new Integer(1);

        public static void main(String args[]) {
          Map m = new HashMap();
          // Inizializza la tabella di frequenza dagli argomenti del main
          for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE : new Integer(freq.intValue() +
                1)));
        }
        System.out.println(m.size()+" distinct words detected:");
        System.out.println(m);
} }
```

C:> java Freq if it is to be it is up to me to delegate

OUTPUT

8 distinct words detected:

{to=3, me=1, delegate=1, it=2, is=2, if=1, be=1, up=1}

Map

- I valori in una mappa sono accessibili per mezzo delle chiavi o anche per mezzo di appositi iteratori:

```
for (Iterator i=m.keySet().iterator(); i.hasNext(); ) {
   System.out.println(i.next());
}
```

*Map

- Set entrySet() – per avere la sequenza di oggetti che rappresentano la coppia chiave valore modellata per mezzo di:

```
public interface Entry {
    Object getKey();
    Object getValue();
    Object setValue(Object value);
}
```

```
for (Iterator i=m.entrySet().iterator(); i.hasNext(); ) {
   Map.Entry e = (Map.Entry) i.next();
   System.out.println(e.getKey() + ": " + e.getValue());
}

//m2 sotto-mappa di m1
if (m1.entrySet().containsAll(m2.entrySet())) {
    ...
}
```

*Funzioni di Utilità

- Collections è una classe con metodi statici che offrono un ampio spettro di funzionalità impiegabili sulle collezioni:
  - boolean addAll(Collection c, T... elements) – per aggiungere elementi ad una data collection c;
  - int binarySearch(List list, Object key[, Comparator c]) – per ricercare un elemento key in una lista, fornendo anche un comparatore;
  - void copy(List dest, List src) – per copiare gli elementi della lista src in quella dest;
  - boolean disjoint(Collection c1, Collection c2) – per verificare che due collection non abbiano elementi in comune;
  - void fill(List list, Object obj) – per sostituire tutti gli elementi in list con obj;
  - int frequency(Collection c, Object o) – per computare le occorrenze nella collection c del valore o;
  - Object max(Collection coll[, Comparator comp]) – per determinare il valore massimo in una collection coll, esiste anche un corrispettivo per il calcolo del minimo.

*Funzioni di Utilità

- boolean replaceAll(List list, Object oldVal, Object newVal) – per sostituire un determinato elemento in list con un nuovo valore;
- void reverse(List list) – per invertire l'ordine degli elementi in list;
- void sort(List list[, Comparator c]) – per ordinare gli elementi in una lista dato esplicitamente, o meno, un criterio di comparazione;
- void swap(List list, int i, int j) – per scambiare di posto gli elementi di posizione i e j in una lista list.