I Contenitori

Introduzione

In Java è disponibile il tipo predefinito ***array***, a cui si aggiungono delle librerie che includono un ragionevole insieme di *classi contenitore* (*container class*).

Tali classi sono spesso chiamate anche *collection* sebbene in Java 2 il nome ***Collection*** sta ad indicare un particolare sottoinsieme di librerie.

Array in Java…

Ci sono due caratteristiche che distinguono gli array dagli altri tipi di contenitori: efficienza e tipo.

L'array è il modo più efficiente che Java fornisce per memorizzare ed accedere direttamente ad una sequenza di oggetti (intesi come riferimenti ad oggetti).

Il prezzo di questa efficienza è quello di avere una dimensione predefinita che **non può essere modificata** durante il proprio ciclo di vita.

Come alternativa, Java fornisce la classe *ArrayList* che però paga la maggiore flessibilità con la minore efficienza.

…Array in Java.

Rispetto ad un contenitore generico, **l'array contiene un tipo specifico di oggetto**.

E' conveniente usare gli array quando si vuole guadagnare in termini di efficienza e controllo sul tipo di oggetti. Negli altri casi, infatti, potrebbero risultare un po' restrittivi.

L'unico membro accessibile di un array è *length* che dice il numero di oggetti contenuti nell'array mentre la sintassi '*[]*' è l'unico modo per accedere all'oggetto di un array.

…Array in Java…

In Java se si superano i confini specificati per una istanza di array (o di classe contenitore), viene generata un eccezione a run-time (***RunTimeException***).

Al contrario, in C++ l'operatore **[ ]** non effettua alcun controllo sui limiti del contenitore.

Contenitori di tipi primitivi…

A differenza degli array, le classi contenitore possono contenere solo *riferimenti ad oggetti*.

Si aggira questo limite ricordando che in Java sono disponibili delle classi "*wrapper*" come *Integer*, *Double* etc. che permettono di inserire dei valori primitivi in un contenitore.

Tuttavia si sottolinea che è più efficiente e semplice creare un array di tipi primitivi anziché un contenitore di classi "wrapper".

Restituzione di un array

In C/C++ non è possibile restituire un intero array ma solo un puntatore all'array (con tutti i problemi del caso dovuti al fenomeno di *memory leak*).

Java utilizza lo stesso approccio restituendo però un array di riferimenti agli oggetti senza alcuna preoccupazione sulla dimensione.

La classe *Arrays*…

Java include nella sua libreria *java.util*, la classe *Arrays* che contiene un insieme di metodi statici che supportano le funzioni di utilità per gli array.

```
public class Arrays {
  // Suppresses default constructor,
  // ensuring non-instantiability.
    private Arrays() {
    }
```

Non ci sono altri costruttori, quindi tutti i metodi definiti devono essere di classe per poter garantire l'utilizzabilità di Arrays.

La classe *Arrays*…

Ci sono quattro funzioni di base:

- *equals()* : confronta per uguaglianza due array;
- *fill()* : riempie un array con un valore;
- *sort()* : ordina un array;
- *binarySearch()* : ritrova un elemento in array ordinato.

Tutti questi metodi sono 'overloaded' per tutti i tipi primitivi e gli oggetti di classe *Object*.

Ai metodi elencati prima si aggiunge *asList()* che prende degli array e li inserisce in un altro contenitore chiamato *List*.

Copia di un array…

La libreria standard di Java fornisce un metodo **static,** *System.arraycopy()*, che effettua la copia di un array più velocemente del classico ciclo *for*. Questo metodo è "overloaded" per trattare tutti i tipi.

Gli argomenti del metodo *arraycopy()* sono :

- l'array sorgente;
- l'indirizzo dell'array sorgente da cui iniziare la copia;
- l'array di destinazione;
- l'indirizzo dell'array di destinazione dove inizia la copia;
- il numero di elementi da copiare.

…Copia di un array.

Se si violano i confini dell'array, verrà sollevata una eccezione.

Nel seguente esempio si mostra come possono essere copiati sia tipi primitivi che oggetti.

```
// Esempio di uso di System.arraycopy()
import com.prova.util.*;
import java.util.*;
 public class CopyingArrays {
public static void main(String[] args) {
 int[] i = new int[25];
 int[] j = new int[25];
 Arrays.fill(i, 47);
 Arrays.fill(j, 99);
 System.arraycopy(i, 0, j, 0, i.length);
 int[] k = new int[10];
 Arrays.fill(k, 103);
 System.arraycopy(i, 0, k, 0, k.length);
 Arrays.fill(k, 103);
 System.arraycopy(k, 0, i, 0, k.length);
 Integer[] u = new Integer[10];
 Integer[] v = new Integer[5];
 Arrays.fill(u, new Integer(47));
 Arrays.fill(v, new Integer(99));
 System.arraycopy(v, 0,u, u.length/2, v.length);
 }
}
```

…Copia di un array.

Si ricorda che nel caso degli oggetti sono duplicati solo i riferimenti agli oggetti, non c'è alcuna duplicazione degli oggetti stessi (*shallow copy*).

Come fare la deep copy? (Usare la clonazione)

Confronto di array

La classe *Arrays* fornisce il metodo *equals()* 'overloaded' per tutti i tipi primitivi e per gli oggetti *Object*. Questo metodo confronta i corrispondenti elementi di due array e restituisce *true* se i due array sono uguali.

La classe *Arrays*…

```
public static boolean equals(long[] a, long[] a2) {
        if (a==a2)
            return true;
        if (a==null || a2==null)
            return false;
        int length = a.length;
        if (a2.length != length)
            return false;
        for (int i=0; i<length; i++)
            if (a[i] != a2[i])
                return false;
        return true;
    }
public static boolean equals(char[] a, char[] a2) {
…
```

La classe *Arrays*…

```
public static boolean equals(Object[] a, Object[] a2) {
        if (a==a2)
            return true;
        if (a==null || a2==null)
            return false;
        int length = a.length;
        if (a2.length != length)
            return false;
        for (int i=0; i<length; i++) {
            Object o1 = a[i];
            Object o2 = a2[i];
            if (!(o1==null ? o2==null : o1.equals(o2)))
                return false;
        }
        return true;
    }
```

Confronto di array

Il confronto di uguaglianza per array di oggetti si basa sui contenuti (mediante Object.equals()).

Il confronto “==” tra oggetti sarebbe “superficiale”.

La classe Object implementa equals nel modo più restrittivo (x==y), mentre la classe String lo ridefinisce in modo da effettuare un confronto fra stringhe. Il polimorfismo del metodo fa il resto.

```
// Uso di Arrays.equals()
import java.util.*;
public class ComparingArrays {
public static void main(String[] args) {
 int[] a1 = new int[10];
 int[] a2 = new int[10];
 Arrays.fill(a1, 47);
 Arrays.fill(a2, 47);
 System.out.println(Arrays.equals(a1, a2));
 a2[3] = 11;
 System.out.println(Arrays.equals(a1, a2));
 String[] s1 = new String[5];
 Arrays.fill(s1, "Hi");
 String[] s2 = {"Hi", "Hi", "Hi", "Hi", "Hi"};
 System.out.println(Arrays.equals(s1, s2));
 }
}
```

Output

true

false

true

Esercizio: Data la classe

```
class Index {
      private int i;
      Index (int i) {this.i=i;}
      int getI(){return i;}
      public boolean equals (Object o){
            return ((Index)o).getI()==this.i;
      }
}
```

Scrivere una classe dotata di un main che dichiari ed inizializzi due array con 5 riferimenti ad oggetti Index e verifichi se i due array contengono nelle stesse posizioni oggetti con lo stesso stato

```
class MainTest{
      public static void main (String args[])
      {
            Index i1[]=new Index[5];
            Index i2[]=new Index[5];
            for(int i=0;i<i1.length;i++)
            {
                  i1[i]=new Index(i);
                  i2[i]=new Index(i);
            }

      System.out.println(Arrays.equals(i1,i2));
      }
}
```

Comparazione degli elementi di un array…

L'approccio di Java è basato sulla tecnica del *callback*, per cui la parte del codice che varia da caso a caso è incapsulata all'interno della rispettiva classe mentre la parte di codice che resta invariata richiamerà il codice che cambia.

Nell'ordinamento:

Parte invariante: algoritmo di ordinamento (a bolle, quicksort, mergsort, etc.). Incluso nella classe Arrays.

Parte variabile: confronto di due istanze di una classe. Incluso nel codice della classe di appartenenza.

Parte invariante…

```
public static void sort(Object[] a) {
        Object aux[] = (Object[])a.clone();
        mergeSort(aux, a, 0, a.length);
private  static  void  mergeSort(Object  src[],  Object
   dest[], int low, int high) {
   int length = high - low;

   // Bubblesort on smallest arrays
   if (length < 7) {
       for (int i=low; i<high; i++)
       for (int j=i; j>low &&
            (dest[j-1] GREATER THAN dest[j]); j--)
           swap(dest, j, j-1);
       return;
   }
```

…

Parte invariante…

```
…
    // Recursively sort halves of dest into src
    int mid = (low + high)/2;
    mergeSort(dest, src, low, mid);
    mergeSort(dest, src, mid, high);
    // If list is already sorted, just copy from src to
    // dest.  This is an optimization.
    if ((src[mid-1] LESS THAN OR EQUAL TO src[mid]) {
       System.arraycopy(src, low, dest, low, length);
       return;
    }
    // Merge sorted halves (now in src) into dest
    for(int i = low, p = low, q = mid; i < high; i++){
        if (q>=high || p<mid && (src[p] LESS THAN OR EQUAL
  TO src[q]) dest[i] = src[p++];
        else
             dest[i] = src[q++];
    }
}
```

…Comparazione degli elementi di un array…

In Java 2 ci sono due modi per offrire funzionalità di confronto.

Il primo è attraverso il *metodo di confronto naturale* che viene comunicato alla classe mediante la implementazione dell'interfaccia *java.lang.Comparable*.

Questa semplice interfaccia è fornita di un singolo metodo,

```
public int compareTo(Object o)
```

che prende un oggetto generico *Object* come argomento e restituisce un valore negativo se l'argomento è più piccolo dell'oggetto corrente, zero se è uguale e un valore positivo se è maggiore.

```
private static void mergeSort(Object src[], Object dest[], int low, int high) {
int length = high - low;
// Insertion sort on smallest arrays
if (length < 7) {
          for (int i=low; i<high; i++)
             for (int j=i; j>low &&(Comparable)dest[j-1]).compareTo((Comparable)dest[j]) >0; j--)
                    swap(dest, j, j-1);
           return;
}
// Recursively sort halves of dest into src
int mid = (low + high)/2;
mergeSort(dest, src, low, mid);
mergeSort(dest, src, mid, high);

// If list is already sorted, just copy from src to dest.  This is an
// optimization that results in faster sorts for nearly ordered lists.
if (((Comparable)src[mid-1]).compareTo((Comparable)src[mid]) <= 0) {
       System.arraycopy(src, low, dest, low, length);
       return;
}
// Merge sorted halves (now in src) into dest
for(int i = low, p = low, q = mid; i < high; i++){
    if (q>=high || p<mid && ((Comparable)src[p]).compareTo(src[q])<=0)
           dest[i] = src[p++];
    else
           dest[i] = src[q++];
    }
}
```

```
// Implementare l'interfaccia Comparable.
import com.prova.util.*;
import java.util.*;
public class CompType implements Comparable {
          int i;  int j;
          public CompType(int n1, int n2) {i = n1;j = n2;}
          public String toString() {return "[i = " + i + ", j = " + j + "]";}
          public int compareTo(Object rv) {
                    int rvi = ((CompType)rv).i;
                    return (i < rvi ? -1 : (i == rvi ? 0 : 1));
          }
 }
 public class MainClass
          public static void main(String[] args) {
          CompType[] a = new CompType[5];
          for (int i=0;i<a.length;i++)
                    a[i]=new CompType((int) (Math.random()*a.length), (int)
          (Math.random()*a.length));
          Arrays.sort(a);
          }
}
```

Esempio

```
public class TestSort{
   public static void main(String args[]){
   Studente s[]=new Studente[5];
   s[0]=new Studente(3,12,”Rossi”, “Paolo”);
   s[1]=new Studente(2,16,”Rossi”, “Mario”);
   System.out.println(s[1].compareTo(s[0]); // stampa -1
   System.out.println(s[0].compareTo(s[1]); // stampa 1

   s[2]=new Studente(1,12,”Verde”, “Paolo”);
   s[3]=new Studente(4,15,”Bianchi”, “Maria”);
   s[4]=new Studente(5,16,”Rossi”, “Maria”);
   Arrays.sort(s);
   Arrays.sort(s,new StudenteComp());
   Arrays.sort(s, new StudenteComp2());
   }
}
```

```
class Studente implements Comparable{
    int matricola;
    int numeroEsami;
    String cognome;
    String nome;
    Studente(int matricola,int numeroEsami, String cognome,String nome){
          this.matricola=matricola;
          this.numeroEsami=numeroEsami;
          this.cognome=cognome;
          this.nome=nome;
    }
    public int compareTo(Object o){
          Studente s=(Studente)o;
          if(s.numeroEsami==numeroEsami){
                    return 0;
          else if(s.numeroEsami<numeroEsami)
                    return -1;
         else return 1;
    }

}
```

Domanda

- Che succede se volessi ordinare s in base al cognome?

```
class StudenteComp implements Comparator
{
   public int compare(Object o1, Object o2){
        Studente s1=(Studente)o1;
        Studente s2=(Studente)o2;
        return s1.cognome.compareTo(s2.cognome)
   }
   public boolean equals(Object obj){
        Studente s1=(Studente)o1;
        Studente s2=(Studente)o2;
        return s1.cognome.equals(s1.cognome);
   }
}
```

Domanda

- Che succede se volessi ordinare s in base alla matricola?

```
class StudenteComp2 implements Comparator
{
   public int compare(Object o1, Object o2){
        Studente s1=(Studente)o1;
        Studente s2=(Studente)o2;
        return (s2.matricola-s1.matricola);
   }
   public boolean equals(Object obj){
        Studente s1=(Studente)o1;
        Studente s2=(Studente)o2;
        return s1.matricola==s2.matricola;
   }
}
```

…Comparazione degli elementi di un array.

Il secondo modo è quello di creare una classe che implementa l'interfaccia ***Comparator*** che ha due metodi:

```
public int compare(Object o1, Object o2)
public boolean equals(Object obj)
```

Il metodo *compare* deve restituire un valore intero negativo, nullo o positivo a secondo del risultato del confronto.

```
public static void sort(Object[] a, Comparator c) {
        Object aux[] = (Object[])a.clone();
        mergeSort(aux, a, 0, a.length, c);
```

```
private static void mergeSort(Object src[], Object dest[],int low, int high, Comparator c) {
          int length = high - low;
          // Insertion sort on smallest arrays
          if (length < 7) {
              for (int i=low; i<high; i++)
                    for (int j=i; j>low && c.compare(dest[j-1], dest[j])>0; j--)
                        swap(dest, j, j-1);
              return;
          }
    // Recursively sort halves of dest into src
    int mid = (low + high)/2;
    mergeSort(dest, src, low, mid, c);
    mergeSort(dest, src, mid, high, c);

    // If list is already sorted, just copy
    // from src to dest.
    if (c.compare(src[mid-1], src[mid]) <= 0) {
        System.arraycopy(src, low, dest, low, length);
        return;
    }
    // Merge sorted halves (now in src) into dest
    for(int i = low, p = low, q = mid; i < high; i++) {
        if (q>=high || p<mid && c.compare(src[p], src[q]) <= 0)
            dest[i] = src[p++];
        else
            dest[i] = src[q++];
    }
}
```

```
// Implementazione dell'interfaccia Comparator
import com.prova.util.*;
import java.util.*;
class CompTypeComparator implements Comparator {

public int compare(Object o1, Object o2) {
int j1 = ((CompType)o1).j;
int j2 = ((CompType)o2).j;
return (j1 < j2 ? -1 : (j1 == j2 ? 0 : 1));
}
}
public class ComparatorTest {
public static void main(String[] args) {
CompType[] a = new CompType[5];
for (int i=0;i<a.length;i++)
        a[i]=new CompType((int) (Math.random()*a.length), (int)
        (Math.random()*a.length));
Arrays.sort(a, new CompTypeComparator());
}
}
```

Introduzione ai Contenitori…

I contenitori forniti da Java 2 possiedono tutte le funzionalità che ci si aspetterebbe da una buona libreria in aggiunta ad una maggiore semplicità di apprendimento e uso rispetto alle analoghe librerie C++.

I contenitori di Java 2 si basano su due distinti concetti:

- ***Collection***: un gruppo di *singoli* elementi a cui spesso sono applicati dei vincoli (e.g., un oggetto della classe *List* deve contenere elementi in una particolare sequenza, uno della classe *Set* non può contenere elementi duplicati).
- ***Map***: un gruppo di *coppie* "chiave - oggetto valore".

La distinzione tra le due categorie di contenitori in Java è quindi basata sul numero di elementi contenuti in ogni locazione del contenitore.

…Introduzione ai Contenitori...

La categoria *Collection* contiene un elemento in ogni locazione. Essa include il tipo *List* che contiene un gruppo di elementi in una particolare sequenza e il tipo *Set* che permette solo l'aggiunta di un elemento di un dato tipo.

L'*ArrayList* è un tipo di *List* e *HashSet* è un tipo di *Set*.

Per aggiungere elementi a qualsiasi *Collection* si usa il metodo *add()*.

Per quanto riguarda il riempimento di un contenitore, si può usare il metodo *fill()* anche se solo per il tipo *List*.

…Introduzione ai Contenitori.

La categoria dei contenitori *Map* contiene coppie di chiave-valore come se fosse una piccola tabella di un database relazionale.

Se si ha un contenitore *Map* che associa le regioni italiane con i rispettivi capoluoghi di regione e si vuole conoscere il capoluogo di regione della Puglia vi si accede come se si avesse un array indicizzato (i contenitori *Map* sono chiamati anche *array associativi*).

Per aggiungere degli elementi ad un *Map* si usa il metodo *put()* che acquisisce in input una chiave ed un valore come argomenti.

…Introduzione ai Contenitori…

Un contenitore *Map* può restituire

1. un *Set* contenente tutte le sue chiavi,
2. una *Collection* contenente tutti i suoi valori,
3. un *Set* di coppie.

I contenitori *Map*, come gli array, possono essere facilmente estesi a dimensioni multiple senza l'aggiunta di nuovi concetti ma semplicemente creando un *Map* i cui valori sono a loro volta dei *Map*.

Visualizzare i Contenitori

A differenza degli array, che consentono la stampa di un elemento alla volta in un ciclo for, i contenitori si possono visualizzare con facilità.

```
import java.util.*;
public class PrintingContainers {
static Collection fill(Collection c) {
 c.add("dog");c.add("dog");c.add("cat");
 return c;
 }
static Map fill(Map m) {
 m.put("dog", "Bosco");m.put("dog", "Spot"); m.put("cat", "Rags");
 return m;
 }
public static void main(String[] args) {
 System.out.println(fill(new ArrayList()));
 System.out.println(fill(new HashSet()));
 System.out.println(fill(new HashMap()));
 }
 }
```

Visualizzare i Contenitori

Per l'esempio precedente avevamo il seguente output:

```
[dog, dog, cat]
[cat, dog]
{cat=Rags, dog=Spot}
```

Una istanza di *Collection* è visualizzata con parentesi quadre ed ogni suo elemento è separato da una virgola.

Una istanza di *Map* è visualizzata con parentesi graffe ed ogni elemento è rappresentato da una coppia chiave-valore separati da un segno di uguale.

Svantaggi dei Contenitori...

Quando si usano i contenitori Java si perde l'informazione relativa al tipo di oggetto che si inserisce in essi.

Ciò porta alle seguenti considerazioni:

- non essendoci controllo sul tipo degli oggetti inclusi nel contenitore, non è possibile essere sicuri di avere dei contenitori di oggetti omogenei;
- l'unica cosa che il contenitore conosce è che contiene dei riferimenti ad oggetti, pertanto è necessario effettuare un cast al tipo corretto prima di utilizzarli.

…Svantaggi dei Contenitori.

Se in un contenitore di riferimenti ad oggetti di tipo 'cerchio' si inserisce un oggetto di tipo 'quadrato', nel momento in cui si andranno a manipolare gli oggetti del contenitore trattandoli tutti come 'cerchio' verrà sollevata una eccezione a run-time (precisamente quando si incontrerà l'oggetto 'quadrato‘).

L’identificazione al run-time del tipo è comunque possibile in Java (vedi RTTI).

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

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

```
//CatsAndDogs.java

import java.util.*;
public class CatsAndDogs {
public static void main(String[] args) {
ArrayList cats = new ArrayList();
for(int i = 0; i < 7; i++)
   cats.add(new Cat(i));
// Non ci sono problemi ad aggiungere un
cane a 7
// gatti !!
cats.add(new Dog(7));
for(int i = 0; i < cats.size(); i++)
   ((Cat)cats.get(i)).print();
// Ma la presenza del cane è rilevata al
run-time!
}
}
```

Iteratori…

Un iteratore è un **oggetto** che si muove attraverso una sequenza di oggetti e ne seleziona uno qualsiasi senza che il programmatore sia a conoscenza della struttura sottostante della sequenza.

Ad ogni contenitore si può chiedere, mediante il metodo *iterator()*, di generare un opportuno iteratore, che altro non è che una istanza di una classe che implementa l'interfaccia *Iterator*.

L'interfaccia *Iterator* di Java permette di:

1. restituire il primo elemento della sequenza sulla prima chiamata del metodo *next()*.
2. prendere l'oggetto successivo nella sequenza con l'invocazione del metodo *next()*;
3. verificare se ci sono altri oggetti nella sequenza con *hasNext()*;
4. rimuovere l'ultimo elemento restituito dall'iteratore con il metodo *remove()*;

Esempio di iteratore

```
import java.util.*;
public class CatsAndDogs2 {
public static void main(String[] args) {
   ArrayList cats = new ArrayList();
   for(int i = 0; i < 7; i++)
       cats.add(new Cat(i));
   Iterator e = cats.iterator();
   while(e.hasNext())
       ((Cat)e.next()).print();
}
}
```

Funzionalità di *List*…

List (interfaccia)

Mantiene gli elementi nell'ordine in cui sono inseriti. Fornisce metodi per l'inserimento e l'eliminazione dei suoi elementi.

Una *List* produrrà un *ListIterator* che potrà spostarsi nella lista in entrambe le direzioni al suo interno.

Per specializzare il comportamento di *List* sono disponibili le sottoclassi:

- **ArrayList** È una *List* implementata con un array. Permette un accesso rapido agli elementi ma risulta lento nell'inserire o rimuovere gli elementi al centro del contenitore.

…Funzionalità di *List*.

ListIterator dovrebbe essere utilizzato solo per accedere agli elementi e non per il loro inserimento ed eliminazione.

- **LinkedList** Fornisce un ottimo accesso sequenziale con inserimenti ed eliminazioni poco costosi dal centro delle *List*. Risulta relativamente lenta negli accessi diretti. Supporta i metodi *addFirst()*, *addLast()*, *getFirst()*, *getLast()*, *removeFirst()* e *removeLast()* che gli permettono di essere usata come uno stack o una coda.

Funzionalità di *Set*...

Set (interfaccia)

Set ha esattamente la stessa interfaccia di *Collection* così non abbiamo funzionalità extra come per i tipi *List*.

Un *Set* contiene solo una istanza di ogni valore dell'oggetto. L'interfaccia *Set* non garantisce l'ordine degli oggetti inseriti. L'univocità degli elementi è stabilita mediante il metodo *equals()*.

Classi che implementano l'interfaccia *Set* sono:

- ***HashSet*** È utilizzato per *Set* che richiedono tempi di accesso più brevi. Per gli oggetti si definisce un metodo *hashCode()*.

```
class Studente{
private int matricola ;
private String cognome;
Studente(int matricola, String cognome){
        this.matricola=matricola;
        this.cognome=cognome;
}
public boolean equals (Object o) {
        return ((Studente)o).getMatricola()==matricola;
}

int getMatricola(){return matricola;}

public int hashCode() {
        return new Integer(matricola).hashCode() ;
}

public String toString(){ return cognome;}
}
```

import java.util.HashSet;

```
Class Tester{
public static void main (String ars[]){
HashSet s=new HashSet();
s.add(new Studente(1,”Bianchi”));
s.add(new Studente(2,”Rossi”));
s.add(new Studente(1,”Verde”));
System.out.println(s);
}
}
```

Output
[Bianchi, Rossi]

…Funzionalità di *Set*…

- ***TreeSet*** È un *Set* ordinato attraverso un albero. In tal modo si può estrarre una sequenza ordinata da un *Set*.

```
class Studente implements Comparable{
private int matricola ;
private String cognome;
Studente(int matricola, String cognome){
this.matricola=matricola;
this.cognome=cognome;
}
public int compareTo (Object o) {
return matricola-((Studente)o).getMatricola();
}

int getMatricola(){return matricola;}

public String toString(){ return cognome;}
}
```

import java.util.TreeSet;

```
class Tester{
public static void main (String ars[]){
TreeSet s=new TreeSet();
s.add(new Studente(1,”Bianchi”));
s.add(new Studente(2,”Rossi”));
s.add(new Studente(1,”Verde”));
System.out.println(s);
}
}
```

Output
[Bianchi, Rossi]

…Funzionalità di *Set*…

Pur accettando l'inserimento di valori duplicati, si può notare che quando il contenuto del *Set* viene visualizzato si vede che essi non sono stati considerati.

I *Set* ordinati (di cui il *TreeSet* è l'unico accessibile) aggiungono funzionalità nuove, rispetto ai *Set* comuni, che permettono di gestire l'ordinamento degli elementi inseriti.

E se volessi ordinare s rispetto al cognome pur considerando due studenti duplicati se hanno la stessa matricola?

Funzionalità di *Map*…

L'interfaccia *Map* permette di accedere agli oggetti di un contenitore attraverso un altro oggetto (contrariamente a quanto accadeva negli ArrayList, dove serviva un numero).

A tale scopo abbiamo i seguenti metodi:

- *put (Object key, Object value)*: aggiunge un valore e lo associa con una chiave;
- *get(Object key)*: restituisce il valore corrispondente alla chiave;
- *containsKey(Object key)*, *containsValue(Object value)*: verifica se *Map* contiene un valore o una chiave.

…Funzionalità di *Map*…

La libreria standard di Java include due tipi differenti di *Map*: *HashMap* e *TreeMap*.

Entrambe le classi hanno la stessa interfaccia ma si distinguono per efficienza.

- **HashMap** garantisce elevate prestazioni per la ricerca di una chiave perché è una implementazione basata sulla tabella hash. Infatti ogni oggetto Java può produrre un codice hash (di tipo *int*) attraverso il metodo *hashCode()* definito nella classe *Object*. *HashMap* utilizza tale codice per effettuare una ricerca efficiente sugli elementi.

…Funzionalità di *Map*…

Il metodo *hashCode()* usa per default l'indirizzo dell'oggetto per poter generare il valore intero.

Può quindi accadere che oggetti differenti per indirizzo, ma identici per valore, siano indicizzati in modo diverso.

```
public class ObjectHashCode {
public static void main(String[] args) {
System.out.println((new Object()).hashCode());
System.out.println((new  Object()).hashCode());}
  }
```

Per risolvere il problema si può sovrascrivere il metodo *hashCode()* per la classe di oggetti chiave da memorizzare in una Map. Questo è quello che avviene per *String* o *Integer*.

…Funzionalità di *Map*.

- **TreeMap** Le coppie chiave-valore sono ordinate (attraverso i metodi della interfaccia *Comparable*). *TreeMap* dispone di un metodo *subMap()* che restituisce una porzione dell'albero costruito per mantenere ordinati gli oggetti.

```
class Studente{
private int matricola ;
private String cognome;
Studente(int matricola, String cognome){
          this.matricola=matricola;
          this.cognome=cognome;
}
public boolean equals (Object o) {
          return
((Studente)o).getMatricola()==matricola;
}

int getMatricola(){return matricola;}

public int hashCode() {
 Integer i=new Integer(matricola);
 return i.hashCode();
}

public String toString(){ return cognome;}
}
```

```
class Counter {
private int i = 1;
Counter (int i){this.i=i;}
public String toString() {
return Integer.toString(i);
}
}
```

```
// Semplice esempio di HashMap.

import java.util.*;

class Statistics {
public static void main(String[] args) {
HashMap hm = new HashMap();
hm.put(new Studente(1,"Bianchi"), new Counter(1));
hm.put(new Studente(2,"Rossi"), new Counter(2));
hm.put(new Studente(2,"Verde"), new Counter(3));
hm.put(new Studente(3,"Bianchi"), new Counter(4));
}
}
```

Output

{Bianchi=1, **Rossi=3,** Bianchi=4}

E se volessi usare un TreeMap?

```
class Studente implements Comparable{
private int matricola ;
private String cognome;
Studente(int matricola, String cognome){
          this.matricola=matricola;
          this.cognome=cognome;
}

int getMatricola(){return matricola;}

public int compareTo(Object o) {
   Studente s=(Studente)o;
   Integer i= new Integer(s.matricola);
   Integer m= new Integer(matricola);
   return m.compareTo(i);
}

public String toString(){ return cognome;}
}
```

```
// Semplice esempio di HashMap.

import java.util.*;

class Statistics {
public static void main(String[] args) {
TreeMap tm = new TreeMap();
tm.put(new Studente(1,"Bianchi"), new Counter(1));
tm.put(new Studente(2,"Rossi"), new Counter(2));
tm.put(new Studente(2,"Verde"), new Counter(3));
tm.put(new Studente(3,"Bianchi"), new Counter(4));
}
}
```