Programmazione Java

Davide Di Ruscio
Dipartimento di Informatica
Università degli Studi dell'Aquila

diruscio@di.univaq.it

Sommario

- Introduzione alle collezioni
- Interfacce
- Implementazione
- Collection
- List
- Set
- Map
- Ordinamento
  - Comparable
  - Comparator
- SortedSet e SortedMap

Collezione (1)

» Una collezione (chiamata anche *container*) è un oggetto che raggruppa elementi multipli in una singola unità

» Sono utilizzate per memorizzare, recuperare e manipolare dati, per trasmetterli da un metodo ad un altro

» Tipicamente rappresentano dati correlati tra loro, come una collezione di numeri telefonici, collezione di lettere, ecc

» Sono state introdotte a partire dalla release 1.2 (*collection framework*)

Collezione (2)

» Un framework per le collezioni è composto in genere da
- **Interfacce**
  - Tipi di dato astratti che rappresentano le collezioni
  - Permettono di manipolare le collezioni indipendentemente dai dettagli della rappresentazione
  - In genere formano una gerarchia
- **Implementazioni**
  - Implementazioni concrete delle interfacce
  - Sono le *strutture dati riusabili*
- **Algoritmi**
  - Metodi che effettuano delle computazioni sulle collezioni, come ad esempio ordinamento, ricerca, …
  - Gli algoritmi sono *polimorfici* poiché gli stessi metodi possono essere applicati a differenti implementazioni
  - Sono le *funzionalità riusabili*

Collezione (3)

» Benefici

- Riduce lo sforzo di programmazione
- Incrementa la velocità e qualità dello sviluppo
- Permette l'interoperabilità tra API non in relazione
- Riduce il tempo di apprendimento e l'utilizzo di nuove API
- Riduce il tempo per lo sviluppo di nuove API
- Aumenta il riuso di software

» Svantaggi

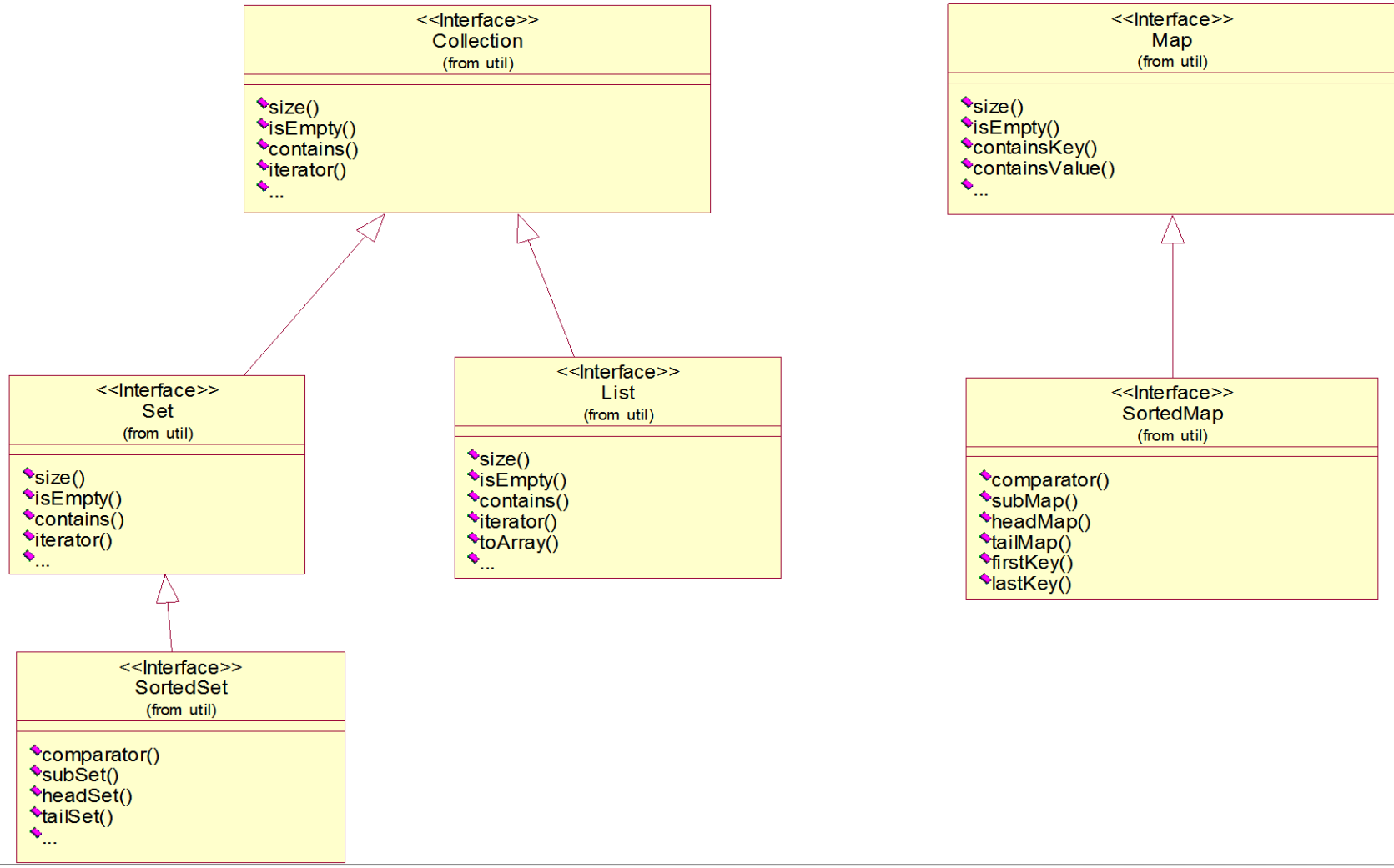
- Generalmente sono abbastanza complesse
- Collection di Java sono abbastanza semplici

Collezione (4)

- In Java si hanno due concetti distinti:
  - Collection: un gruppo di singoli elementi
  - Map: un gruppo di coppie di oggetti chiave-valore

- La distinzione su basa sul numero degli elementi contenuti in ciascuna posizione del contenitore

Interfacce (1)



Interfacce (2)

» Collection

- Root della gerarchia
- Rappresenta un gruppo di oggetti conosciuti come *elementi*
- E' il minimo comun denominatore che tutte le collezioni implementano
- Alcune implementazioni
  - Ammettono duplicati altre no
  - Ordinamento su elementi oppure no
- JDK non ha implementazioni di tale interfaccia
- Vengono fornite implementazioni delle sue sotto-interfacce come `Set`, `List`

Interfacce (3)

- **Set**
  - Collezione che **non** può contenere duplicati
  - Astrazione dell'*insieme* matematico
- **List**
  - Collezione ordinata (detta anche *sequenza*)
  - Può contenere elementi duplicati
  - Si accede agli elementi mediante un indice intero (*posizione*)
- **Map**
  - Oggetto che mappa una chiave ad un valore
  - Non possono contenere chiavi duplicate ovvero una chiave mappa un solo valore

Interfacce (4)

» **SortedSet**

– Insieme dove gli elementi sono ordinati in ordine ascendente

– Operazioni aggiuntive per utilizzare l'ordinamento

» **SortedMap**

– Map dove le chiavi sono ordinate in ordine ascendente

» L'ordine per `SortedSet` e le `SortedMap` viene stabilito all'atto dell'utilizzo di un'implementazione (`Comparator`) oppure dando l'ordine agli oggetti che contengono (`Comparable`)

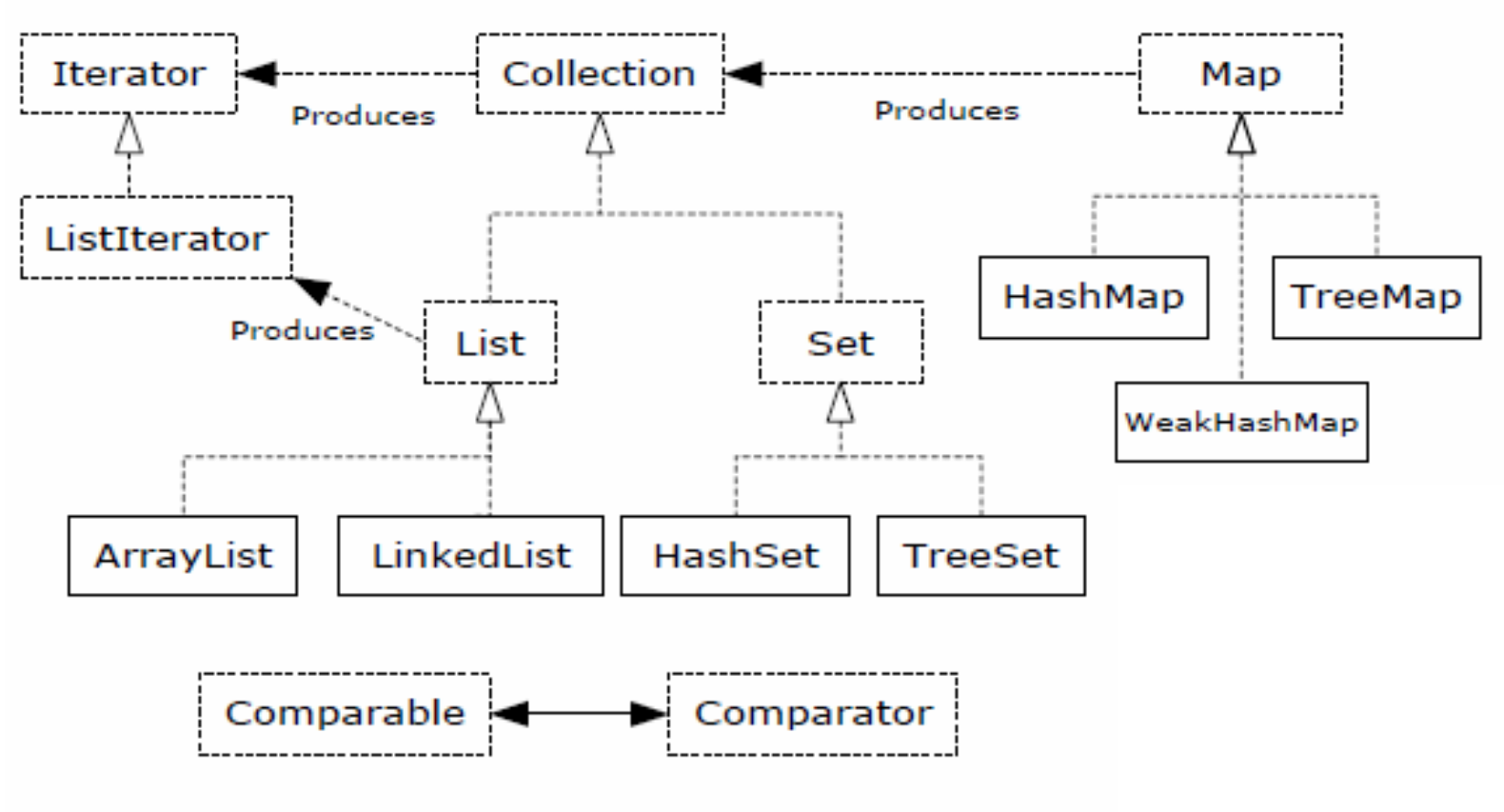
Implementazione (1)

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

Implementazione (2)

- Sono presenti almeno due implementazioni per ogni interfaccia
- Implementazioni primarie
  - `HashSet`
  - `ArrayList`
  - `HashMap`
- `TreeSet` e `TreeMap` implementano `SortedSet` e `SortedMap`
- `Vector` e `Hashtable` erano presenti prima dell'introduzione delle collection
  - Modificate per implementare le nuove interfacce
- Tutte le implementazioni sono serializzabili (`java.io.Serializable`) e supportano il metodo `clone()`

Implementazione (3)



Implementazione (4)

- `Set`
  - `HashSet` è molto più veloce di `TreeSet`
    - tempo costante vs. tempo logaritmico
  - `HashSet` non garantisce l'ordinamento
  - `TreeSet` sì
  - `HashSet` necessità della capacità iniziale che ha impatto su performance
    - Default 101 che è sufficiente
    - Altrimenti costruttore appropriato
    - Vedere Documentazione

Implementazione (5)

» `List`

- `ArrayList` è più veloce di `LinkedList` poiché permette un accesso posizionale con tempo costante e non deve allocare un oggetto `Node` per ogni elemento nella Lista
- Se vengono aggiunti frequentemente elementi all'inizio della lista oppure viene iterata la lista eliminando degli elementi allora conviene utilizzare `LinkedList` poiché vengono eseguite in tempo costante
- `ArrayList` ha un parametro iniziale (*capacità iniziale*) che identifica la dimensione iniziale dell'array utilizzato per memorizzare gli elementi
- `LinkedList` non ha alcun parametro

» Implementazioni di `Map` uguali a quelle di `Set`

Esempio


```
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()));
  }
}
```

(Vedere PrintingContainers.java)

Collection (1)


```
public interface Collection {

  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(Object element); // Opt
  boolean remove(Object element); // Opt
  Iterator iterator();

  boolean containsAll(Collection c);
  boolean addAll(Collection c); // Opt
  boolean removeAll(Collection c); // Opt
  boolean retainAll(Collection c); // Opt
  void clear(); // Opt

  Object[] toArray();
  Object[] toArray(Object a[]);

}
```

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

Collection (2)

» Per convenzione tutte le implementazioni hanno un costruttore con argomento una `Collection` che inizializza la nuova collezione con gli elementi di quella specificata

» Esempio

– Supponiamo di avere una `Collection c` (che può essere un Set oppure una List)

– `List l = new ArrayList(c);`

Collection (3)

» Esempio 1

```
public class Collections1 {

    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 ) array[ i ];
            System.out.println( "Elemento di array:" + element );
        }
        String[] array1 = ( String[] ) c.toArray();  //ClassCastException  (vedere
  implementazione per esempio di AbstractCollection)
        String[] str = ( String[] ) c.toArray( new String[ 0 ] );
        for ( int i = 0; i < str.length; i++ ) {
            String element = str[ i ];
            System.out.println( "Elemento di str: " + element );
        }
    }
}
```

Vedere Collections1.java

Collection (4)

» Esempio 2

```
public class Collections2 {
    public static void main( String[] args ) {
        Collection c = new ArrayList();
        c.add( "ten" );
        c.add( "eleven" );

        for ( Iterator i = c.iterator(); i.hasNext(); ) {
            String element = ( String ) i.next();
            System.out.println( "Elemento i-esimo:" + element );
            if ( element.equals( "eleven" ) ) {
                i.remove();
            }
        }
        System.out.println( "Numero Elementi: " + c.size() );
    }
}
```

Vedere Collections2.java

Collection (5)

- 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
- E' possibile contenere tipi eterogenei
- Non è possibile inserire valori di tipi primitivi
  - E' necessario utilizzare tipi wrapper `Integer, Long`
- Java 5 risolve tali problemi utilizzando i **generics**

Collection (6)

» Esempio

```
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);
  }
}
```

Collection (7)


```
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));
    // Not a problem to add a dog to cats:
    cats.add(new Dog(7));

    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.get(i)).id();
      // Dog is detected only at run time
  }

}
```

Vedere CatsAndDogs.java

Svantaggio delle collections (1)

- Senza l'uso di ulteriori caratteristiche, si perde l'informazione sul tipo dell'oggetto che viene inserito in un contenitore
  - Un contenitore colleziona riferimenti Object che è la radice di tutte le classi e può in tal modo collezionare oggetti di qualsiasi tipo
- In particolare:
  - poiché le informazioni sul tipo vengono eliminate quando si inserisce il riferimento di un oggetto di un contenitore, non esistono limitazioni sul tipo di oggetto che può essere inserito nel contenitore;
  - Poiché le informazioni sul tipo vengono perse, la sola cosa che il contenitore sa è che contiene un riferimento a un oggetto. E' quindi necessario eseguire un cast al tipo corretto prima di utilizzarlo.

Svantaggio delle collections (2)

» Per ovviare a questo problema è possibile:

- Usare i *generics* introdotti in Java 1.5

  (vedere CatsAndDogsGenerics.java)

- Creare nuovi tipi basati su collection

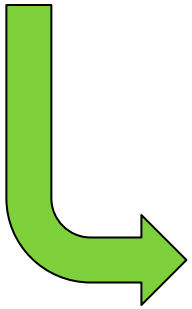
  (vedere MouseListTest.java)

Iteratore

- Un iteratore è un oggetto il cui compito è scorrere una sequenza di oggetti e selezionare ciascun oggetto nella sequenza senza che il programmatore conosca né si occupi della struttura sottostante a tale sequenza
- Dato un contenitore è possibile ottenere l'iteratore corrispondente invocando il metodo `iterator()`
- L'oggetto che viene tornato è di tipo `Iterator` che fornisce i seguenti metodi:
  - `next()` per acquisire l'oggetto successivo nella sequenza
  - `hasNext()` per verificare se esistono altri oggetti nella sequenza
  - `remove()` per eliminare l'ultimo elemento restituito dall'iteratore

Esempio


```
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));
    // Not a problem to add a dog to cats:
    cats.add(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.get(i)).id();
      // Dog is detected only at run time
  }
}
```



```
public class CatsAndDogs2 {

  public static void main(String[] args) {
    List 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()).id();
  }
}
```

Vedere Printer.java,
Hamster.java, HamsterMaze.java

List (1)


```
public interface List extends Collection {
  //Positional Access
  Object get(int index);
  Object set(int index, Object element);

  //Optional
  void add(int index, Object element);
  Object remove(int index);
  boolean addAll(int index, Collection c);

  // Search
  int indexOf(Object o);
  int lastIndexOf(Object o);

  // Iteration
  ListIterator listIterator();
  ListIterator listIterator(int index);

  // Range-view
  List subList(int from, int to);
}
```

List (2)


```
interface ListIterator extends Iterator {
  boolean hasNext();
  Object next();
  boolean hasPrevious();
  Object previous();
  int nextIndex();
  int previousIndex();

  // Optional
  void remove();
  void set(Object o);
  void add(Object o);
}
```

From Iterator

From Iterator

List (3)

» Possono esserci elementi duplicati
» Operazioni aggiuntive
  – Accesso posizionale (si parte da 0) ovvero manipolazione degli elementi in base alla posizione nella lista
  – Ricerca di un determinato oggetto e ritorno della posizione numerica
  – Sotto-liste
  – Implementazioni `ArrayList`, `LinkedList` e `Vector`
    • `LinkedList` fornisce un ottimo accesso sequenziale, con veloci inserimenti ed eliminazioni in arbitrarie posizioni di una `List`. Relativamente lento per l'accesso diretto.
    • `ArrayList` consente un rapido accesso agli elementi, ma è lento nelle operazioni di inserimento e rimozione di elementi in una qualsiasi posizione delle lista
  – `add` e `addAll` aggiungono i nuovi elementi alla fine
» Due Liste sono uguali se gli elementi sono gli stessi nello stesso ordine

List (4)

» Esempio 1

```
public class TestList {
    public static void main( String[] args ) {
        List list = new ArrayList();
        boolean b;
        Object o;
        int i;
        list.add( 1, "x" );                    // Add at location 1
        list.add( "x" );                       // Add at end
        b = list.contains( "1" );              // Is it in there?
        // Lists allow random access, which is cheap
        // for ArrayList, expensive for LinkedList:
        o = list.get( 1 );                     // Get object at location 1
        i = list.indexOf( "1" );               // Tell index of object
        b = list.isEmpty();                    // Any elements inside?
        i = list.lastIndexOf( "1" );           // Last match
        list.remove( 1 );                      // Remove location 1
        list.remove( "3" );                    // Remove this object
        list.set( 1, "y" );                    // Set location 1 to "y"

        i = list.size();                       // How big is it?
        list.clear();                          // Remove all elements
    }
}
```

Vedere TestList.java

List (5)

» Esempio 2

```
import java.util.*;
public class Shuffle {
  public static void main(String args[]) {
       List l = new ArrayList();
       for (int i=0; i<args.length; i++)
            l.add(args[i]);
       Collections.shuffle(l, new Random());
       System.out.println(l);
  }
}
```

Vedere Shuffle.java

List (6)

» Esempio 3

```
import java.util.*;
public class Shuffle2 {
  public static void main(String args[]) {
       List l = Arrays.asList(args);
       Collections.shuffle(l);
       System.out.println(l);
  }
}
```

Vedere Shuffle2.java

List (7)

» Esempio completo

```
import java.util.*;
public class Deal {
  public static void main(String args[]) {
        int numHands = Integer.parseInt(args[0]);
        // Make a normal 52-card deck
        int cardsPerHand = Integer.parseInt(args[1]);
        String[] suit = new String[] {"spades", "hearts",
                                      "diamonds", "clubs"};
        String[] rank = new String[] {"ace","2","3","4","5",
                                      "6","7","8",”9","10",
                                      "jack","queen","king"};
        List deck = new ArrayList();
        for (int i=0; i<suit.length; i++)
                for (int j=0; j<rank.length; j++)
                        deck.add(rank[j] + " of " + suit[i]);
        Collections.shuffle(deck);
        for (int i=0; i<numHands; i++)
                System.out.println(dealHand(deck, cardsPerHand));
  }
```

.....

Vedere Deal.java

List (8)

.....

```
  public static List dealHand(List deck, int n) {
        int deckSize = deck.size();
        List handView = deck.subList(deckSize-n, deckSize);
        List hand = new ArrayList(handView);
        handView.clear();
        return hand;
    }
}
```

```
C:> java Deal 4 5
```

OUTPUT

```
[8 of hearts, jack of spades, 3 of spades, 4 of spades, king of diamonds]
[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts, queen of hearts]
[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds, 9 of clubs]
[8 of spades, 6 of diamonds, ace of spades, 3 of hearts, ace of hearts]
```

Vedere Deal.java

Creare uno stack con una LinkedList

- Uno stack viene spesso identificato con il termine contenitore LIFO
- L'oggetto LinkedList dispone di metodi che implementano direttamente la funzionalità di uno stak
- Tuttavia una class stack può rappresentare meglio la situazione

Vedere Stack.java

Creare una coda con una LinkedList

- Una coda è un contenitore di tipo FIFO
  - L'ordine di inserimento sarà uguale a quello di estrazione
- LinkedList dispone di metodi per supportare il comportamento di tipo coda e questi possono essere utilizzati in una classe Queue

Vedere Queue.java

Set (1)


```
public interface Set extends Collection {

  // Basic Operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(Object element); //Opt
  boolean remove(Object element); //Opt
  Iterator iterator();

  // Bulk Operations
  boolean containsAll(Collection c);
  boolean addAll(Collection c); //Opt
  boolean removeAll(Collection c); //Opt
  boolean retainAll(Collection c); //Opt
  void clear(); //Opt

  // Array Operations
  Object[] toArray();
  Object[] toArray(Object a[]);

}
```

Set (2)

- Stessi metodi dell'interfaccia `Collection`
- Sono proibiti elementi `e1` ed `e2` tali che `e1.equals(e2)` ed almeno uno è `null` (ovvero duplicati)
- Implementazioni `HashSet TreeSet LinkedHashSet`
  - `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
- Esempio
  - Supponiamo di avere una `Collection c`
  - `Collection noDups = new HashSet(c);`
  - Vengono eliminati i duplicati

Set (3)

» Esempio 1

```
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]
```

NOTA: Modificando `HashSet` in `TreeSet` si ottiene l'ordinamento

Vedere FindDups.java

Set (4)

» Esempio 2

```
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]);
        uniques.removeAll(dups);
        // Destructive set-difference
       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]
```

Vedere FindDups2.java, Set1.java

Map (1)


```
public interface Map {

  // Basic Operations
  Object put(Object key, Object value);
  Object get(Object key);
  Object remove(Object key);
  boolean containsKey(Object key);
  boolean containsValue(Object value);
  int size(); boolean isEmpty();

  // Bulk Operations
  void putAll(Map t);
  void clear();

  // Collection Views public
  Set keySet();
  public Collection values();
  public Set entrySet();
```

……

Map (2)

.....

```
  //Interface for entrySet elements
  public interface Entry {
       Object getKey();
       Object getValue();
       Object setValue(Object value);
  }

}
```

Map (3)

- Oggetto che mappa chiavi a valori
- Non può contenere duplicati delle chiavi
- Implementazioni `HashMap`, `TreeMap`, `Hashtable`
  - `HashMap` da preferire rispetto ad `Hashtable`. L'implementazione è basata su una tabella hash e 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
- Per convenzione tutte le implementazioni hanno un costruttore con argomento una `Map` che inizializza la nuova mappa con gli elementi di quella specificata
- Esempio
  - Supponiamo di avere una `Map m`
  - `Map m1 = new HashMap(m);`

Map (4)

» Esempio 1

```
import java.util.HashMap;
import java.util.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( "key" ) );
    }
}
```

Vedere TestMap.java

Map (5)

» Esempio 2

```
public class Freq {
  private static final Integer ONE = new Integer(1);
  public static void main(String args[]) {
        Map m = new HashMap();
        // Initialize frequency table from command line
        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}
```

Vedere Freq.java

Map (6)

» E’ possibile ottenere delle collezioni dalla Mappa

- `Set keySet()`
  - Insieme delle chiavi contenute nella Mappa
- `Collection values()`
  - Collezione dei valori contenuti nella Mappa
  - Non è un insieme (`Set`) poiché la mappa può contenere valori multipli ovvero più valori associati alla stessa chiave
- `Set entrySet()`
  - L’insieme delle coppie chiave-valore contenute nella mappa
  - Elemento dell’insieme `Map.Entry`

Map (7)

» Esempi

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

 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())) {
    ...
 }
```

Ordinamento (1)

» `List l` può essere ordinata nel seguente modo
  – `Collections.sort(l)`
    • Algoritmo utilizzato: variante merge sort
  – Se gli elementi della lista sono
    • `String` ordinamento lessicografico
    • `Date` ordinamento cronologico
    • ……
  – Ma come è possibile?
    • `String` e `Date` **implementano** l'interfaccia `Comparable`

```
public interface Comparable {
    public int compareTo(Object o);
}
```

    • Viene lanciata un'eccezione (`ClassCastException`) se gli elementi non implementano `Comparable` utilizzando il metodo `sort()`

Ordinamento (2)

Classi di Java core che implementano `Comparable`

Byte	Numerico con segno
Character	Numerico senza segno
Long	Numerico con segno
Integer	Numerico con segno
Short	Numerico con segno
Double	Numerico con segno
Float	Numerico con segno
BigInteger	Numerico con segno
BigDecimal	Numerico con segno
File	Lessicografico sul path name (dipende dal S.O.)
String	Lessicografico
Date	Cronologico
CollationKey	Lessicografico locale-specific

Ordinamento (3)


```
public class Name implements Comparable {
    private String  firstName, lastName;

    public Name(String firstName, String lastName) {
        if (firstName==null || lastName==null)
            throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String firstName()    {return firstName;}
    public String lastName()     {return lastName;}

    public boolean equals(Object o) {
        if (this == o ) return true;
        if (!(o instanceof Name))
            return false;
        Name n = (Name)o;
        return n.firstName.equals(firstName) &&
               n.lastName.equals(lastName);
    }
```

Vedere Name.java

Ordinamento (4)


```
.....
  public int hashCode() {
        return 31 * firstName.hashCode() +
                    lastName.hashCode();
  }

  public String toString() {
         return firstName + " " + lastName;
  }

  public int compareTo(Object o) {
        Name n = (Name)o;
        int lastCmp = lastName.compareTo(n.lastName);
        return (lastCmp!=0 ?
                          lastCmp :
                          firstName.compareTo(n.firstName));
  }
}
```

Vedere Name.java

Set2.java

Ordinamento (5)

Esempio

```
import java.util.*;
class NameSort {
    public static void main(String args[]) {
        Name n[] = {
            new Name("John", "Lennon"),
            new Name("Karl", "Marx"),
            new Name("Groucho", "Marx"),
            new Name("Oscar", "Grouch")
        };
        List l = Arrays.asList(n);
        Collections.sort(l);
        System.out.println(l);
    }
}
```

```
C:> java NameSort
```

OUTPUT

```
[Oscar Grouch, John Lennon, Groucho Marx, Karl Marx]
```

Ordinamento (6)

» Interfaccia: `Comparator`

```
public interface Comparator {
      int compare(Object o1, Object o2);
}
```

» Maggior controllo sull'ordinamento

» Permette l'ordinamento su oggetti che non implementano l'interfaccia `Comparable`

» E' possibile effettuare diversi ordinamenti mediante tale interfaccia

» E' simile all'interfaccia `Comparable` soltanto che il confronto viene effettuato al di fuori della classe

(Vedi AlphabeticComparator.java, Utilities)

SortedSet (1)

- `SortedSet` è un `Set` dove i suoi elementi sono mantenuti in ordine ascendente utilizzando l'interfaccia `Comparable` oppure `Comparator` (fornita all'atto di creazione)
- Per convenzione le implementazioni vengono fornite di un costruttore che ha come argomento
  - Una `Collection`
  - Un `SortedSet`
  - Un `Comparator`
  - Un `Comparator` e un `SortedSet`

SortedSet (2)


```
interface SortedSet extends Set {
   // Range-view
   SortedSet subSet(Object fromElement, Object toElement);
   SortedSet headSet(Object toElement);
   SortedSet tailSet(Object fromElement);

   // Endpoints
   Object first();
   Object last();

   // Comparator access
   Comparator comparator();
}
```

» `subSet()` estremo sx escluso ed estremo dx incluso
» `headSet()` dall'inizio fino all'elemento specificato escluso
» `tailSet()` dall'elemento specificato fino alla fine

SortedMap (1)

- `SortedMap` è un `Map` dove i suoi elementi sono mantenuti in ordine ascendente utilizzando l'interfaccia `Comparable` oppure `Comparator` (fornita all'atto di creazione)
- Per convenzione le implementazioni vengono fornite di un costruttore che ha come argomento
  - Una `Collection`
  - Un `SortedMap`
  - Un `Comparator`
  - Un `Comparator` e un `SortedMap`

SortedMap (2)


```
interface SortedMap extends Map {
    Comparator comparator();

    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);

    Object firstKey();
    Object lastKey();

}
```

» `subMap(), headMap(), tailMap()` come `SortedSet`

Altri metodi di Collections

- Ricerca binaria

```
public static int binarySearch(List list, Object key);
public static int binarySearch(List list,
                               Object key, Comparator c);
```

- Shuffle: *randomizza* la collection

```
public static void shuffle(List list)
public static void shuffle(List list, Random rnd);
```

- Other

```
public static void reverse(List list);
public static void fill(List list, Object obj);
public static void copy(List src, List dest);
public static Object min(Collection c);
public static Object max(Collection c);
```

Operazioni non supportate

- Talvolta i metodi definiti nell'interfaccia Collection non "funzionano"
- Non esiste un modo per specificare "l'opzionalità" dell'implementazione di un metodo di un'interfaccia
- La chiamata a un metodo non supportato genererà una UnsopportedOperationException
  - Questa soluzione impedisce un'esplosione di interfacce
  - Non è nemmeno possibile catturare tutti i casi speciali in più interfacce, poiché qualcuno può sempre definirne una nuova

(Vedere Unsopported.java)