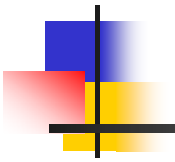


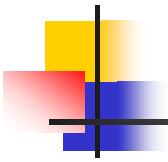

Fondamenti di Informatica L-B
Esercitazione n°6
Java: Collezioni, Classe Wrapper & Generics



A.A. 2005/06

Tutor: Loris Cancellieri

loris.cancellieri@studio.unibo.it

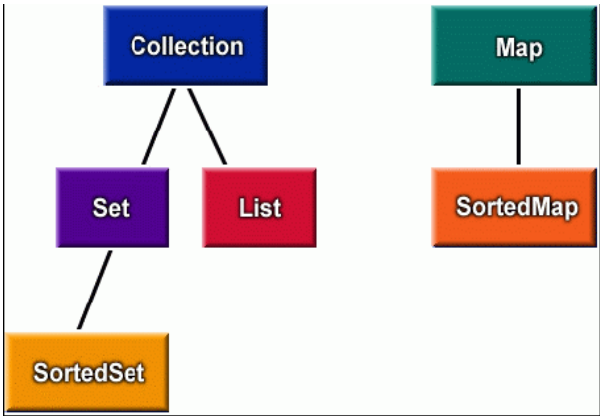
Strutture Dati in Java

- **Molto spesso, una computazione si basa su una o più strutture dati, di vario tipo:**
  - **insiemi, code, stack, tabelle, liste, alberi…**
- **Data la loro importanza, Java ne offre un'ampia scelta nella *Java Collection Framework (JCF)***
  - **interfacce che definiscono *insiemi di funzionalità***
  - **classi che ne forniscono *varie implementazioni***
- **Sono *contenitori "generici" per oggetti***
  - **fino a JDK 1.4, si usava Object come tipo generico**
  - **da Java 1.5 in poi: supporto ai TIPI GENERICI**

Java Collection Framework (package java.util)

Interfacce Fondamentali:

- **Collection**: nessuna ipotesi su elementi duplicati o relazioni d'ordine
- **List**: introduce l'idea di sequenza
- **Set**: introduce l'idea di insieme di elementi quindi senza duplicati
- **SortedSet**: Insieme ordinato
- **Map**: introduce il concetto di mappa e cioè di insieme che associa chiavi (identificatori univoci) a valori



Java Collection Framework Quadro generale

		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

Implementazioni fondamentali

- **per le liste: *ArrayList, LinkedList, Vector***
- **per le tabelle hash: *HashMap, HashSet, Hashtable***
- **per gli alberi: *TreeSet, TreeMap***

Altri concetti di utilità:

- **Concetto (interfaccia) di *iteratore***
- **Classe factory *Collections***

Java Collection Framework
Quadro generale

		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

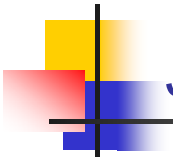
Due parole sulle classi di implementazione:

- **le *linked list* sono liste realizzate con puntatori**
- **i *resizable array* sono liste di array**
- **i *balanced tree* sono alberi realizzati con puntatori**
- **le *tabelle hash* sono tabelle realizzate tramite *funzioni hash,* ossia funzioni che associano una entità a un valore tramite una funzione matematica.**

Java Collection Framework
Come era e come è cambiata

Nella gestione delle collezioni in java possiamo trovare un taglio netto fra la gestione delle collezioni nelle versioni precedenti il java 5 e il java 5 stesso.

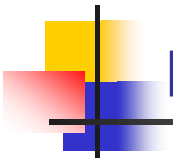
Nelle versioni precedenti parleremo di versione «classica» del Java Collection Framework, con il java 5 parleremo di Java Collection Framework con i tipi Generici



Java Collection Framework «classico»

- **Fino a Java 1.4, si usava il tipo generico Object come mezzo per ottenere contenitori generici**
  - **i metodi che aggiungono / tolgono *oggetti* dalle collezioni prevedono un parametro di tipo Object**
  - **i metodi che cercano / restituiscono *oggetti* dalle collezioni prevedono un valore di ritorno Object**

- **Conseguenza:**
  - **si possono aggiungere/togliere alle/dalle collezioni oggetti di qualunque tipo, *TRANNE i tipi primitivi***
  - **questi ultimi devono prima essere rispettivamente incapsulati in un oggetto (BOXING) / estratti da un oggetto che li racchiuda (UNBOXING)**



Problema con i tipi primitivi:

I tipi primitivi sono i "mattoni elementari" del linguaggio, ma *non sono oggetti* e pertanto *non possono essere inseriti in strutture dati che si aspettano degli Object,* come ad esempio:

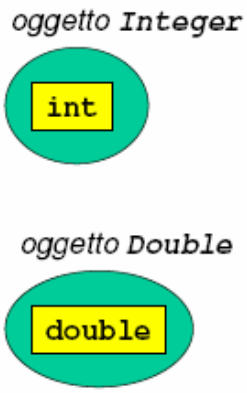
```
Object put(Object key, Object value); // Map
```

Da qui l'esigenza di *incapsulare i valori primitivi in oggetti:* a tal fine Java definisce delle classi particolari: le *classi WRAPPER*

Ogni classe wrapper ha un *nome (quasi) identico al tipo primitivo incapsulato*, con l'iniziale maiuscola**.**

Tipi Primitivi e Classi Wrapper

Tipo primitivo	Classe "wrapper" corrispondente
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double



Ogni classe wrapper definisce:

- un costruttore che accetta il valore primitivo da incapsulare
- un metodo che restituisce il valore primitivo incapsulato

Tipi Primitivi e Classi Wrapper

- **Per incapsulare un valore primitivo *(BOXING)***
  - `Integer i = new Integer(`*valore int*`)`
  - `Double d = new Double(`*valore double*`)`
  - `Character c = new Character(`*valore char*`)`
  - `...`
- **Per estrarre il valore incapsulato *(UNBOXING):***
  - **`Integer`** fornisce il metodo **`intValue()`**
  - **`Double`** fornisce il metodo **`doubleValue()`**
  - **`Character`** fornisce il metodo **`charValue()`**
  - `...`

ESEMPI

```
Integer i = new Integer(33);   // boxing
int k = i.intValue();          // unboxing
```

Classi Wrapper e funzionalità:

Le classi Wrapper non hanno operatori aritmetici:

```
new Integer(3) + new Integer(2);
```

NO!

Quindi, per operare su due valori incapsulati in oggetti wrapper, in linea di principio occorre:

- estrarre i valori incapsulati (unboxing)
- svolgere le operazioni su essi,
- costruire un oggetto per il valore risultato (boxing)

```
Integer z =
 new Integer(x.intValue() + y.intValue());
```


Java 1.5: boxing automatico

In Java 1.5, boxing e unboxing di valori primitivi diventano *automatici.*

Quindi, ora si può scrivere:

```
Integer x = new Integer(8);
Integer y = new Integer(4);
Integer z = x + y;
```

OK

perché la conversione da Integer a int degli operandi (unboxing) e la successiva conversione del risultato in Integer (boxing) sono automatiche.

Java 1.5: boxing automatico

Analogamente è quindi possibile inserire valori primitivi in strutture dati che accettano Object:

```
List l = new ArrayList();
l.add(10); // OK in Java 1.5
```

perché viene attuata automaticamente la conversione da int a Integer (boxing) del valore primitivo 10.

Fino a Java 1.4.2, avremmo invece dovuto scrivere:

```
l.add(new Integer(10));
```

facendo quindi esplicitamente l'operazione di boxing.

Java Collection Framework
Esempi

		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

Inizieremo con un esempio sul set, che come sapete evoca il concetto di insieme di elementi

Esempio: Set

Il codice presentato analizza le parole elencate dalla linea di comando, e stampa:

- le parole duplicate
- il numero di parole distinte
- la lista delle parole depurata dei duplicati

```
import java.util.*;
public class TrovaDupl {
   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("Parola duplicata: " + args[i]);
   System.out.println(s.size() + " parole distinte: "+s);
   }
}
```

Scegliamo una delle possibili implemetazioni

Il metodo add restituisce false se l'elemento esiste già

Esempio: Set

Notare che, escluso il momento delle creazione:

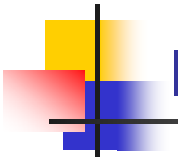
- ***si usa sempre il tipo dell'interfaccia (Set)***
- **NON il tipo della classe scelta come implementazione.**

Eseguendolo si ottiene:

```
F:\temp>java TrovaDupl ciao Marco sono Mario ciao Maria sono Marco
Parola duplicata: ciao
Parola duplicata: sono
Parola duplicata: Marco
5 parole distinte: [Mario, Marco, ciao, Maria, sono]
```

Lo stesso risultato si sarebbe ottenuto usando anche un'altra implementazione: ad esempio:

```
Set s = new TreeSet();
```



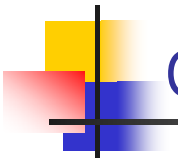
Iteratori

Un *iteratore* è una entità capace di *garantire l'attraversamento di una collezione* con una semantica chiara e ben definita *(anche se la collezione venisse modificata)*

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

Di fatto, un iteratore offre un metodo **`next()`** per esplorare uno ad uno gli elementi della collezione: il metodo **`hasNext()`** permette di sapere quando non ci sono più elementi.

Per ottenere un iteratore su una qualsiasi collezione, basta chiamare l'apposito metodo **`iterator()`**



Come si usano?

Ad esempio per elencare tutti gli elementi di una collezione, la cosa migliore è procurarsi un iteratore e ciclare sfruttando questo:

```
. . .
Iterator i;
for (i = s.iterator();i.hasNext(); ) {
System.out.println(i.next() + " ");
}
```

Come già detto, per ottenere un iteratore su una qualsiasi collezione basta **chiamare l'apposito metodo `iterator()`**, usando **`hasNext()`** per controllare il ciclo e **`next()`** per avanzare

Eseguendolo si ha:

```
import java.util.*;
public class TrovaDupl {
    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("Parola duplicata: " + args[i]);
    //System.out.println(s.size() + " parole distinte: "+s);
    System.out.println("Stampo Lista");
    Iterator i;
    for (i = s.iterator();i.hasNext(); ) {
        System.out.print(i.next() + " ");
        }

    }
}
```

```
F:\temp>java TrovaDupl ciao Marco sono Mario ciao Maria sono Marco
Parola duplicata: ciao
Parola duplicata: sono
Parola duplicata: Marco
Stampo Lista
Mario Marco ciao Maria sono
```


Iteratori: nuovo costrutto for

- **Il concetto di *iteratore* come modo per *attraversare una collezione di elementi* rende conveniente definire una nuova forma del costrutto for**
- **Anziché esplicitare una variabile di controllo, si esprime l'idea di visitare uno ad uno tutti gli elementi di una collezione: Java realizzerà questo procurandosi un iteratore e gestendolo *automaticamente*.**

```
for (Object x : qualsiasicollezione) {
  // x denota via via gli elementi della collezione
}
```

Applicandolo all'esempio precedente:

```
for (Object x : s) {
      System.out.print(x);
}
```

Da Set a List

List introduce un concetto di *sequenza*: definisce un tipo di contenitore che può contenere duplicati.

```
public interface List extends Collection {
  Object get(int index);
  Object set(int index, Object element); // Optional
  void   add(int index, Object element); // Optional
  Object remove(int index);              // Optional
  abstract boolean addAll(int index, Collection c); //Optional
  int indexOf(Object o);
  int lastIndexOf(Object o);
  ListIterator listIterator();
  ListIterator listIterator(int index);
  List subList(int from, int to);
}
```

La lista è una sequenza
→ c'è una idea di posizione

Nuovo iteratore per liste: ha il concetto di *posizione*
→ può iniziare da un indice dato

Implementazioni di List

- **Fino a JDK 1.4, la forma di lista più usata era Vector**
- **Da Java 1.5, l'uso dell'interfaccia List costituisce la scelta primaria**
  - **metodi con nomi più brevi, parametri in ordine più naturale**
  - **varie implementazioni disponibili (ArrayList, LinkedList)**
  - **anche Vector (reingegnerizzato) ora implementa List!**

Da Iterator a ListIterator

Poiché List introduce un concetto di *sequenza*, nasce anche un nuovo tipo di iteratore che la sfrutta.

```
public interface ListIterator extends Iterator {
   boolean hasNext();
   Object next();
   boolean hasPrevious();
   Object previous();
   int nextIndex();
   int previousIndex();
   void remove();          // Optional
   void set(Object o);     // Optional
   void add(Object o);     // Optional
}
```

La lista è una sequenza navigabile anche a ritroso

L'iteratore di lista ha un concetto di indice prossimo o precedente

```
ListIterator listIterator();
ListIterator listIterator(int index);
```

Si può farsi dare un iteratore che inizi da un indice specificato

Esempio Liste (1)

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

public class EsempioListe {
    public static void main (String args[]) {
    Contatore[] w;
    w = new Contatore[4];

    for (int i=0;i<w.length;i++) {
    w[i] = new Contatore(i,10*i);
    w[i].incrementa();
    }

    //Creo una Lista di Contatore.
    List l1 = Arrays.asList(w);

    ListIterator it1;
```

Esempio Liste (2)

```
//Stampo il vettore di contatori:
    for (int i=0;i<w.length;i++) {
    System.out.println("Il contatore n "+i+" vale "+w[i].leggi());
    }

    //Stampo la Lista con iteratore
    for (it1=l1.listIterator();it1.hasNext(); ) {
        System.out.print("Contatore vale "+((Contatore)it1.next()).leggi()+
   "\n");
        }

    Contatore c1 = new Contatore (6, 12);
    //l1.add(c1); Questo non lo posso fare poichè l1 è ancora un riferimento
   a qualcosa di astratto, non punta ad una implementazione concreta
```


Esempio Liste (3)

```
List l2 = new ArrayList();

    //Copio l1 n l2
    for (it1=l1.listIterator();it1.hasNext(); ) {
        l2.add((Contatore)it1.next());
        }

    l2.add(c1);

    System.out.println("Ho aggiunto c1 a l2 e c1 ora vale 6");

    for (it1=l2.listIterator();it1.hasNext(); ) {
        System.out.print("Contatore vale "+((Contatore)it1.next()).leggi()+
    "\n");
        }
```

Esempio Liste (4)

```
Contatore c2 = new Contatore (10, 20);
     l2.add(c2);
     System.out.println("Ho aggiunto c2");

     //Ora vado a cercare la posizione di c1:
     System.out.println("La lista contiene "+l2.size()+" elementi");
     System.out.println("Elemento c1 in posizione "+l2.lastIndexOf(c1));

     //Se voglio incrementare c1 posso farlo:
     int pos = l2.lastIndexOf(c1);
     ((Contatore)l2.get(pos)).incrementa();

     //Ristampo tutto
     for (it1=l2.listIterator();it1.hasNext(); ) {
        System.out.print("Contatore vale "+((Contatore)it1.next()).leggi()+
  "\n");
        }
```


Esempio Liste (5)

```
//Aggiungo nella posizione da me desiderata
     Contatore c3 = new Contatore (30,50);
     l2.add(2,c3);
     System.out.println("Elemento c1 in posizione
  "+l2.lastIndexOf(c1));
     System.out.println("Elemento c3 in posizione
  "+l2.lastIndexOf(c3));

     System.out.println("Ristampo di nuovo la lista");
     //Ristampo tutto
     for (it1=l2.listIterator();it1.hasNext(); ) {
        System.out.print("Contatore vale
  "+((Contatore)it1.next()).leggi()+ "\n");
        }
```

Esempio Liste (6)

```
//Se io voglio recuperare l'indice dell'elemento il cui contatore
  vale 3, se ne ho più di uno recupero solo il primo?
     int indice = -1;
     int valContToFind = 3;
     for (it1=l2.listIterator();it1.hasNext(); ) {
       if ((((Contatore)it1.next()).leggi())==valContToFind) {
       indice = it1.nextIndex()-1;
       break;
       }
     }
     if (indice==(-1)) {
     System.out.println("Non ho trovato un contatore che vale
  "+valContToFind+" nella lista");
     }else { System.out.println("Il primo contatore nella lista che
  vale "+valContToFind+" è in posizione "+indice);
     }
```


Esempio Liste (7)

```
  Collections.sort(l2);
      //Ristampo tutto
      for (it1=l2.listIterator();it1.hasNext(); ) {
        System.out.print("Contatore vale
    "+((Contatore)it1.next()).leggi()+ "\n");
        }     }
      }//End Main
}
```

Altro Esempio che sfrutta Comparable

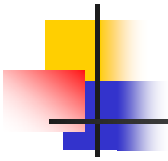
```
class Persona implements Comparable {
private String nome, cognome;
public Persona(String nome, String cognome) {
this.nome = nome; this.cognome = cognome;
}
   public String nome() {return nome;}
   public String cognome() {return cognome;}
   public String toString() {return nome + " " + cognome;}
   public int compareTo(Object x) {
        Persona p = (Persona) x;
        int confrontoCognomi = cognome.compareTo(p.cognome);
        return (confrontoCognomi!=0 ? confrontoCognomi :
   nome.compareTo(p.nome));
}
}
```


Altro Esempio che sfrutta Comparable

```
class NameSort {
public static void main(String args[]) {
Persona elencoPersone[] = {
  new Persona("Eugenio", "Bennato"),
  new Persona("Roberto", "Benigni"),
  new Persona("Edoardo", "Bennato"),
  new Persona("Eros", "Ramazzotti")
};
List l = Arrays.asList(elencoPersone);
Collections.sort(l);
System.out.println(l);
}
}
```

Produce una List a partire dall'array dato

Ordina tale List in senso ascendente



Problemi del JCF Classico

- **Usare *il tipo generico Object* per definire *contenitori generici* causa *non pochi problemi***
  - **equivale ad abolire il controllo di tipo!**
  - **rende possibili *operazioni sintatticamente corrette* ma *semanticamente errate***
  - **determina quindi il rischio di errori a runtime pur a fronte di compilazioni formalmente corrette**
  - **la correttezza è affidata a "commenti d'uso!!**
  - **rende il linguaggio non "type safe“**
- **Java 1.5 introduce un nuovo approccio, basato sul concetto di *TIPO PARAMETRICO ("generici")***

Vediamolo prima in un esempio:

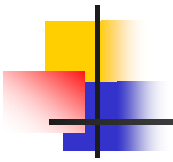
```
//JDK 1.4.x
List lista = new ArrayList();
lista.add("ciao");
lista.add("sono una lista");
String s1 = (String)lista.get(0);
String s2 = (String)lista.get(1);
...
StringBuffer sb = (StringBuffer)lista.get(0);
```

Per la versione 1.4 il codice viene compilato, ma genera un errore a runtime

Vediamolo prima in un esempio (2):

```
//1.5
List<String> lista = new ArrayList<String>();
lista.add("ciao");
lista.add("sono una lista");
//...
String s1 = lista.get(0);
String s2 = lista.get(1);
...
StringBuffer sb = lista.get(0);
```

Per la versione 1.5 il codice non viene neanche compilato



Il nuovo approccio:

- **È sbagliato abolire il controllo di tipo!**
- **Occorre *un altro modo* per esprimere genericità, che consenta un controllo di tipo a compile time**
  - **type safety: *"se si compila, è certamente corretto"***
- **Java 1.5 introduce i tipi parametrici ("generici")**
- **Il tipo può essere un parametro:**
  - **in funzioni statiche, in classi e metodi di classi**
  - **notazione <TIPO>**
- **Si possono definire relazioni fra "tipi generici"**
  - **recupera il "lato buono" dell'ereditarietà fra collezioni, inquadrandolo in un contesto solido**

Esempio Liste con Generics

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

public class EsempioListeGenerics {
    public static void main (String args[]) {
    Contatore[] w;
    w = new Contatore[4];

    for (int i=0;i<w.length;i++) {
    w[i] = new Contatore(i,10*i);
    w[i].incrementa();
    }

    List<Contatore> l1 = new ArrayList<Contatore>();

    l1.add(w[0]);   l1.add(w[1]);  l1.add(w[2]); l1.add(w[3]);
```


Esempio Liste con Generics(2)

```
ListIterator<Contatore> it1;

    //Stampo la Lista con iteratore
    System.out.println("Stampo la lista con gli iteratori");
    for (it1=l1.listIterator();it1.hasNext(); ) {
        System.out.print("Contatore vale "+(it1.next()).leggi()+ "\n");
        }

     Contatore c1 = new Contatore (10, 20);
     l1.add(1,c1);

     //Ora vado a cercare la posizione di c1:
     System.out.println("La lista contiene "+l1.size()+" elementi");

     System.out.println("Elemento c1 in posizione "+l1.lastIndexOf(c1));
```

Esempio Liste con Generics(3)

```
//Se io voglio recuperare l'indice dell'elemento il cui contatore vale
   3, se ne ho più di uno recupero solo il primo...
     int indice = -1;
     int valContToFind = 3;
     Contatore c2 = new Contatore(100,150);


//Trova il contatore il cui valore vale valContToFind ne estrae l'indice
   per restituirmelo poi lo cancella dalla lista e ci inserisce il nuovo
   contatore
     for (it1=l1.listIterator();it1.hasNext(); ) {
        if (((it1.next()).leggi())==valContToFind) {
        indice = it1.nextIndex()-1;
        Contatore c3 = it1.previous();
        it1.remove();
        it1.add(c2);
        break;
        }
     }
```


Esempio Liste con Generics(4)

```
     if (indice==(-1)) {
     System.out.println("Non ho trovato un contatore che vale
   "+valContToFind+" nella lista");
     }else {
     System.out.println("Il primo contatore nella lista che vale
   "+valContToFind+" è in posizione "+indice);
     }


     //Ordino i contatori.....
     Collections.sort(l1);


     //Ristampo tutto
     for (it1=l1.listIterator();it1.hasNext(); ) {
        System.out.print("Contatore vale "+(it1.next()).leggi()+ "\n");
        }
     }//End Main
}
```

Modifiche in contatore per usare i generics e rendere tutto type-safe:

```
public class Contatore implements Comparable<Contatore>

   public int compareTo(Contatore c) { return valore-
   c.valore; }
   public boolean equals(Contatore c) {
         return (valore == c.valore); }

   public int compareTo(Object o) { return
   compareTo((Contatore)o); }
   public boolean equals(Object o) { return
      equals((Contatore) o); }
```