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

A.A. 2007/08

Tutor: Barbara Pettazzoni

barbara.pettazzoni@studio.unibo.it

Strutture dati avanzate in Java

- Come abbiamo visto la scorsa volta, gli array in Java son sí piú potenti che nel C, ma restano limitati: non possiamo incrementare la dimensione senza perdere valore informativo, per esempio.
- Java fornisce un'ampia scelta di strutture dati (array dinamici, pile, alberi…) all'interno del Java Collection Framework :
  - Definisce una serie di interfacce che specificano un insieme di funzionalitá
  - Fornisce giá *diverse implementazioni*
- Queste strutture dati sono contenitori "generici" per gli oggetti
  - Fino alla versione di Java 1.4, si utilizzava la classe Object
  - Da Java 5 in poi, é stato aggiunto il supporto ai *tipi generici!!*

2

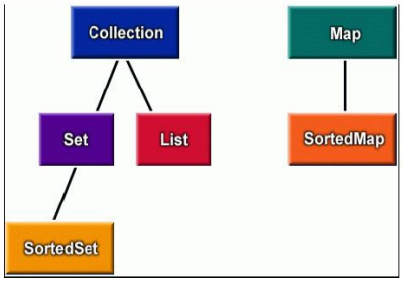
Il package java.util

- Avete giá avuto modo di utilizzare parte di questo package, grazie allo StringTokenizer.
- Le interfacce fondamentali sono:

1. **Collection**: non indica nessuna ipotesi su elementi duplicati o relazioni d'ordine
2. **List**: introduce l'idea di sequenza
3. **Set**: introduce l'idea di insieme senza duplicati
4. **SortedSet**: si tratta di un insieme senza elementi duplicati e ordinati
5. **Map**: introduce il concetto di mappa, cioé di un insieme che associa a delle chiavi (che sono univoche) dei valori.

3

Gerarchia ed implementazioni



		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

4

Quadro generale

Implementazioni fondamentali:

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

Per poter lavorare correttamente con queste strutture, sará necessario aver bene in mente:

- Cosa é un **iteratore**
- A cosa puó servire la classe factory ***Collections***

Quadro generale (2)

		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

- Le **linked list** sono liste realizzate mediante puntatori
- I **resizable array** sono liste di array
- I **balanced tree** sono alberi realizzati mediante puntatori
- Le **tabelle hash** sono tabelle realizzate mediante delle *funzioni hash*, che sono delle particolari funzioni matematiche in grado di associare ad un'entitá un valore numerico univoco.

java.util, versione classica

- Fino a Java 1.4 (la versione "classica"), queste strutture dati utilizzavano il tipo generico `Object` per poter aggiungere/togliere/inserire gli oggetti. Infatti, se andiamo a dare un'occhiata alla Javadoc 1.4 dell' `ArrayList` :

boolean	add(Object o) Appends the specified element to the end of this list.
Object	get(int index) Returns the element at the specified position in this list.
Object	remove(int index) Removes the element at the specified position in this list.

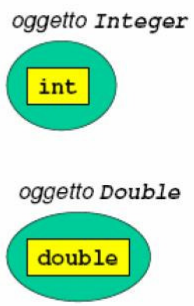
- Ma quindi…come si poteva fare con i tipi primitivi? **Non sono oggetti!!** Qualche idea?

Le classi wrapper (1)

- I tipi primitivi sono i "mattoni elementari" del linguaggio, ma *non sono oggetti*, e quindi *non possiamo usarli in quei metodi in cui si richiede un oggetto generico!*
- Fino a Java 1.4, era necessario eseguire il **boxing** (ovvero, l'incapsulamento di un tipo primitivo in un oggetto) e l'**unboxing** (l'operazione duale, dall'oggetto si vuole il tipo primitivo) a manina, utilizzando delle apposite classi, dette **wrapper**
- Vi é una classe wrapper *per ogni tipo primitivo* : il nome é "quasi" uguale al nome del tipo primitivo.

Le classi wrapper (2)

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, a cui si passa il tipo primitivo
- Un metodo che permette di ottenere il tipo primitivo contenuto nell'oggetto

Le classi wrapper (3)

- Vediamo un semplice esempio di codice:

```
Integer i = new Integer(3);              //BOXING
int valueInt = i.intValue();             //UNBOXING
Character c = new Character('a');        //BOXING
char valueChar = c.charValue();          //UNBOXING
Double d = new Double(2.1);              //BOXING
double valueDouble = i.doubleValue();    //UNBOXING
```

Le classi wrapper (4)

- *Non possiamo eseguire operazioni sugli oggetti incapsulati!*

```
Integer x = new Integer(3);
Integer y = new Integer(2);
Integer z = new Integer();

z = x + y;
```


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

Quindi, ogni volta va fatto prima l'**unboxing**, quindi **si svolge l'operazione desiderata**, e infine si crea il nuovo oggetto, facendo nuovamente **boxing!!!**

11

java.util, versione classica…esempio di codice!

Metto insieme mele, pere, ananas…

```
Vector vettore = new Vector();
vettore.add(new Contatore());
vettore.add("Ciao");
vettore.add(new Integer(3));
vettore.add(new Vector());
…
for(int i=0; i < vettore.size(); i++)
        System.out.println(vettore.elementAt(i));
```

- Il codice…compila? Ovviamente **sí**
- Ma…é del buon codice? **Mica tanto!**

Per esempio, per poter usare i metodi della classe Contatore, dovró fare un **cast** a Contatore…ma ció potrebbe non essere possibile!
Possibile errore a run time!!

12

java.util, da Java 5 in poi

- Da Java 5 in poi, son stati introdotti i **tipi generici** : come si puó anche vedere dalla Javadoc corrispondente, ora si possono costruire delle **strutture dati tipate, a seconda dell'oggetto che si vuole inserire!**

boolean	add(E o) Appends the specified element to the end of this list.
void	add(int index, E element) Inserts the specified element at the specified position in this list.
E	get(int index) Returns the element at the specified position in this list.

- E per i tipi primitivi...é cambiato qualcosa?

13

Le classi wrapper in Java 5

- In Java 5 é stato introdotto il **boxing/unboxing automatico**:

```
Integer x = new Integer(3);
Integer y = new Integer(2);
Integer z = x + y;
```

- Questo codice ora non da piú errore in fase di compilazione! In automatico, Java fa l'unboxing, esegue la somma e crea il nuovo oggetto di tipo Integer!

```
List l = new ArrayList();
l.add(3);
```

- Anche qui...boxing automatico!

14

Esempio di codice in Java 5

```
Vector<Contatore> vettore = new Vector<Contatore>();
vettore.add(new Contatore());
vettore.add("Ciao");
vettore.add(new Integer(3));
vettore.add(new Vector());
…
for(int i=0; i < vettore.size(); i++)
        System.out.println(vettore.elementAt(i));
```

- Il codice…compila? **NO!!!**

Abbiamo costruito un vettore tipato! Il compilatore si accorge che si sta cercando di inserire un oggetto di tipo String, e non prosegue!

java.util…esempi vari

		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

Ora guarderemo un po' di esempi di codice che coinvolgono le diverse strutture del package java.util!

Esempio: Set (1)

- Come giá detto, questa struttura dati impedisce che un oggetto possa essere inserito piú di una volta
- Questo programma riceve come input da linea di comando una serie di parole, e stampa le parole duplicate, il numero di parole distinte e la lista senza duplicati!

```
import java.util.*;
public TestSet
{
        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("Num di parole distinte: " + s.size());
          System.out.println("Lista ottenuta: " + s);
        }
}
```

Il metodo add verifica giá se l'oggetto é presente o meno!

17

Esempio: Set (2)

- Da notare che si é usato ***il tipo dell'interfaccia (Set), NON il tipo della classe.***
- In questo modo, cambiando la classe, il comportamento del programma ***non cambia!***
- Esempio:

```
import java.util.*;
public TestSet
{
        public static void main(String args[])
        {
          Set s = new TreeSet();
          …
```

18

L'interfaccia `Iterator`

- Un ***iteratore*** é un'entitá in grado 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
}
```

- Il metodo `next` permette di esplorare uno ad uno gli elementi della lista, mentre `hasNext` verifica se vi sono altri elementi da visitare
- Il metodo `iterator()` di una qualunque classe di java.util restituisce l'iteratore corrispondente alla lista in uso

19

Come utilizzare gli iteratori

- Nel codice d'esempio sui set, potevamo realizzare noi in questo modo la stampa di tutti gli elementi:

```
…
Iterator iterator = s.iterator();
while(iterator.hasNext())
{
        System.out.print(index.next() + " ");
}
…
```

- Una volta ottenuto l'iteratore, si cicla finche' il metodo `hasNext` restituisce vero, e `next` ritorna l'elemento corrispondente.

20

Il nuovo costrutto for

- Gli iteratori permettono di realizzare una nuova forma di ciclo for, molto piú leggibile; invece che definire una variabile di controllo su cui ciclare, **si esprime l'idea di visitare uno ad uno tutti gli elementi della collezione**: Java prenderá l'iteratore **automaticamente**!
- Riprendiamo sempre il codice di prima:

```
…
for(Object o : s)
{
        System.out.print(o.toString() + " ");
}
…
```

21

Il nuovo costrutto for (2)

- E se avessimo una **collezione tipata**, per esempio sulle stringhe come in questo caso?

```
import java.util.*;
public TestSet
{
        public static void main(String args[])
        {
          Set<String> s = new HashSet<String>();
          …
          for(String string: s)
          {
               System.out.print(s + " ");
          }
        }
}
```

Ancora piú leggibile!! E vi é anche il controllo sul tipo!

22

Da Set a List

- Passiamo quindi a vedere l'interfaccia **List** : le strutture che implementano questa interfaccia definiscono una sequenza di elementi (che possono anche essere 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 é una sequenza, é stato aggiunto il concetto di posizione!

Nuovo tipo di iteratore, con concetto di posizione!

23

Implementazioni dell'interfaccia **List**

- Prima di Java 5, la struttura dati piú usata era la classe **Vector**.
- Da Java 5 in poi invece, l'interfaccia **List** diventa la scelta primaria:
  - Metodi con nomi piú brevi, e parametri in un ordine piú naturale
  - Vi sono varie implementazioni (**ArrayList**, **LinkedList**)
  - **Vector** ha subito pure una reingegnerizzazzione, per cui implementa anche lui questa interfaccia!

24

Da Iterator a ListIterator

- Visto che List introduce il concetto di sequenza, é disponibile un nuovo tipo di iteratore:

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

Si puó navigare la lista a ritroso

L'iteratore ha il concetto di indice successivo e precedente

25

Esempio con List (1)

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

public class EsempioList
{
        public static void main(String args[])
        {
                Contatore [] contatori = new Contatore[4];
                for(int i = 0; i < 4; i++)
                {
                        contatori[i] = new Contatore(i,i*10);
                        contatori[i].incrementa();
                }

                // Creo una lista contatore
                List list = Arrays.asList(contatori);
…
```

26

Esempio con List (2)

```
ListIterator iterator = list.iterator();

//Stampa dal vettore
for(int i=0; i<contatori.length; i++)
        System.out.println("Il contatore " + i +
          "vale: " + contatori[i].leggi());

//Stampa mediante iteratore
while(iterator.hasNext())
{
        Contatore c = (Contatore) iterator.next();
        System.out.println("Contatore vale: " +
           c.leggi);
}
Contatore c1 = new Contatore(6,12);
/* list.add(c1); Non si puó ancora fare, list é un
   riferimento a un qualcosa di ancora astratto,
   non vi é un'implementazione concreta!*/
```

27

Esempio con List (3)

```
List list2 = new ArrayList();
for(iterator = list.iterator(); iterator.hasNext(); )
{
        list2.add((Contatore) iterator.next());
}
list2.add(c1);
System.out.println("Ho aggiunto c1 e vale 6");

iterator = list2.iterator(); //Perché?
while(iterator.hasNext())
{
        Contatore c = (Contatore) iterator.next();
        System.out.println("Contatore vale: " +
           c.leggi());
}
```

28

Esempio con List (4)

```
…
Contatore c2 = new Contatore(10,20);
list2.add(c2);

//Dove si trova c1?
System.out.println("La lista contiene " + list2.size() +
       " contatori");
System.out.println("c1 si trova in posizione: " +
       list2.lastIndexOf(c1));

//Voglio incrementare c1!
int index = list2.lastIndexOf(c1);
((Contatore) list2.get(index)).incrementa();
iterator = list2.iterator(); //Perché?
while(iterator.hasNext())
{
       Contatore c = (Contatore) iterator.next();
       System.out.println("Contatore vale: " +
          c.leggi());
}
```

29

Esempio con List (5)

```
…
  //Voglio aggiungere un contatore
  // in una posizione determinata
  Contatore c3 = new Contatore(30,50);
  list2.add(2,c3);
  System.out.println("c1 si trova in posizione: " +
         list2.lastIndexOf(c1));
  System.out.println("c3 si trova in posizione: " +
         list2.lastIndexOf(c3));

  //Voglio recuperare il primo contatore con valore 3
  int index = -1, valToFind = 3;
  iterator = list2.iterator();
  while(iterator.hasNext())
  {
         Contatore c = (Contatore) iterator.next();
…
```

30

Esempio con List (6)

```
…
        if(c.leggi() == valToFind)
        {
                index = iterator.nextIndex() - 1;
                break; //Uhmmm… recupero sempre e solo il primo?
        }
    } //Fine while
    if(index < 0)
    {
        System.out.println("Non esiste contatore con" +
                "valore: " + valToFind);
    }
    else
    {
         System.out.println("Il primo contatore con" +
                "valore " + valToFind + " si trova " +
                "in posizione: " index);
    }
…
```

31

Esempio con List (7)

```
…
        Collections.sort(list2);
        iterator = list2.iterator();
        while(iterator.hasNext())
        {
                Contatore c = (Contatore) c.next();
                System.out.println("Contatore vale: " +
                        c.leggi());
        }
    } //End main
}
```

32

Ulteriore esempio con Comparable (1)

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

33

Ulteriore esempio con Comparable (2)

```
public class SortNames
{
        public static void main(String args[])
        {
                Persona [] persone = {new Persona("Albert", "Einstein"),
                        new Persona("Eros", "Ramazzotti"),
                        new Persona("Eugenio","Bennato"),
                        new Persona("Amaro", "Ramazzotti")};
                List l = Collections.asList(persone);
                Collections.sort(l);
                System.out.println(l);
        }
}
```

34

Ancora sui tipi generici!

- Le versioni precedenti a Java 5 rendevano possibile *operazioni sintatticamente corrette* (il compilatore non dava errore) ma *semanticamente errate* (avevamo poi degli errori a run time!
- Il controllo sul tipo ci permette di sviluppare del codice **type safe** anche quando utilizziamo delle strutture dati avanzate!
- Conviene quindi tipizzare sempre a compile time le strutture dati (*Esercizio: verificate come si dovrebbero modificare i codici precedenti per renderli type safe*)

35

Esempio `Comparable` type safe

```
public class Persona implements Comparable<Persona>
{
  private String nome, cognome;
  public Persona(String nome, String cognome)
  {
        this.nome = nome;
        this.cognome = cognome;
  }

  public boolean equals(Object o)
  {
        Persona p = (Persona) o;
        return(nome.equals(p.nome) && (cognome.equals(p.cognome));
  }

  public int compareTo(Persona p)
  {
        int val = cognome.compareTo(p.cognome);
        return (if (val != 0)? val : nome.compareTo(p.nome));
  }
}
```

36

Esempio `Comparable` type safe

- Non c'é piú bisogno del metodi `compareTo` generico!
  - I confronti avverranno quindi solo con corrette istanze della classe Persona!
- Tuttavia, non tutti I metodi si possono rendere type-safe: nel caso di `equals`, é necessario mantenere come parametro Object
  - Altrimenti si farebbe *overloading*, e non *overriding* del metodo: il compilatore dovrebbe scegliere a seconda del parametro, quale dei due metodi chiamare
    - Questo significa che, se il tipo dell'oggetto passato non é `Persona`, allora verrá richiamato il metodo originale della classe `Object` → potenziali errori!!
- *Esercizio: riscrivete la classe Contatore, rendendola type safe.*

37

Esercizio finale

- La prestigiosa concessionaria Vend O'Tutt vi ha incaricato di realizzare un servizio innovativo: dovete realizzare un'applicazione mediante la quale un possibile acquirente possa consultare il listino delle auto presenti. Le auto sono caratterizzate da prezzo, chilometraggio, colore e nome. Un utente puó scegliere o di visualizzare tutte le macchine, oppure di specificare un parametro su cui cercare (es, tutte le auto rosse, tutte le auto con un prezzo inferiore a 21000…). La lista delle auto stampate deve essere comunque ordinata nell'ordine prezzo > chilometraggio > colore > nome. Il programma termina solo quando l'utente sceglie l'apposita opzione.
- Sul sito potete trovare il file di testo auto.txt che contiene le auto per l'esercizio
- *Opzionale, solo per i piú ardimentosi:* dopo un po' di tempo, la concessionaria vi richiama chiedendovi di poter inserire anche l'opzione di ordinare con chilometraggio > prezzo > nome > colore. *Suggerimento:* controllate se la classe `Collections` vi puó fornire degli indizi interessanti…

38

Soluzione: la classe Auto (1)

```
public class Auto implements Comparable<Auto>
{
        private String nome, colore;
        private int prezzo, km;

        public Auto(String nome, int prezzo, int km, String colore)
        {
                this.nome = nome;
                this.prezzo = prezzo;
                this.km = km;
                this.colore = colore;
        }
        …
        public boolean equals(Object o)
        {
                Auto auto = (Auto) o;
                return (auto.nome.equals(nome) &&
                        auto.colore.equals(colore) &&
                        auto.prezzo == prezzo && auto.km == km);
        }
…
```

39

Soluzione: la classe Auto (2)

```
  public int compareTo(Auto auto){
       int compare1 = prezzo - auto.prezzo;
       if(compare1 != 0)
               return compare1;
       compare1 = km - auto.km;
       if(compare1 != 0)
               return compare1;
       compare1 = colore.compareTo(auto.colore);
       if(compare1 != 0)
               return compare1;
       return nome.compareTo(auto.nome);
  }
  public String toString(){
       return "(Nome: " + nome + ", Prezzo: " + prezzo + ", KM: " + km + ",
  colore: " + colore + ")";}
}
```

40

Soluzione: il main (1)

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

public class Concessionaria
{
   private static Lettore lettore;
   public static void main(String args[])
   {
        //Leggiamo il file con le macchine, e
        //memorizziamo le auto in un'apposita
        //struttura ordinata.
        try
        {
                List<Auto> listaAuto = new Vector<Auto>();
                Lettore leggiFile = new Lettore(new
                      FileReader("auto.txt"));
```

41

Soluzione: il main (2)

```
…
while(!leggiFile.eof())
{
   String linea = leggiFile.leggiLinea();
   StringTokenizer tokenizer = new StringTokenizer(linea," ");
   String nome = tokenizer.nextToken();
   int prezzo = Integer.valueOf(tokenizer.nextToken());
   int km = Integer.valueOf(tokenizer.nextToken());
   String colore = tokenizer.nextToken();
   listaAuto.add(new Auto(nome,prezzo,km,colore));
}
Collections.sort(listaAuto);

//A questo punto, possiamo preparare l'interfaccia
lettore = new Lettore();
boolean fine = false;
System.out.println("Benvenuti presso la Vend O'Tutt!");
```

42

Soluzione: il main (3)

```
…
while(!fine)
{
   stampaInterfaccia();
   String richiesta = lettore.leggiLinea();
   try
   {
        int richiestaNum = Integer.parseInt(richiesta);
        List<Auto> ricerca;
        switch(richiestaNum)
        {
                case 1: stampaAuto(listaAuto);
                         break;
                case 2: System.out.print("Inserisci il prezzo massimo: ");
                         try
                         {
```

43

Soluzione: il main (4)

```
…
  int prezzoMassimo = Integer.parseInt(lettore.leggiLinea());
  ricerca = new Vector<Auto>();
  for(Auto auto : listaAuto)
  {
    if(auto.getPrezzo() <= prezzoMassimo)
         ricerca.add(auto);
  }
  stampaAuto(ricerca);
}
catch(Exception e)
{System.out.println("Valore errato di prezzo"); }
break;
//Le altre opzioni sono simili tranne la 6, che mette a true fine
// l'opzione di default, che stampa un messaggio di errore
```

44

Soluzione: stampaInterfaccia

```
private static void stampaInterfaccia()
{
   System.out.println("Per favore, indica la scelta desiderata fra le
         seguenti: ");
   System.out.println("1) Stampa tutte le auto");
   System.out.println("2) Auto inferiori a un certo prezzo");
   System.out.println("3) Auto inferiori a un certo chilometraggio");
   System.out.println("4) Auto con un certo colore");
   System.out.println("5) Auto di un certo tipo");
   System.out.println("6) Esci");
   System.out.print("Indica la tua scelta (inserisci il numero): ");
}
```

45

Soluzione: stampaAuto

```
private static void stampaAuto(List<Auto> l)
{
   System.out.println("------------");
   if(l.size() > 0)
   {
        for(Auto auto : l)
        {
                System.out.println(auto);
        }
   }
   else
        System.out.println("Non esistono auto con il criterio indicato");
   System.out.println("------------");
}
```

46

Soluzione ardimentosa

- Se avete letto la javadoc, avrete notato che esiste anche un altro modo per ordinare mediante **Collections**: si puó infatti utilizzare una classe che implementi l'interfaccia **Comparator**
- In questo modo, la classe Collections ordinerá secondo l'ordinamento specificato! Si potrebbe pensare quindi di realizzare piú classi che implementino questa interfaccia, in grado di ordinare la lista a seconda dell'esigenza dell'utente.

47

La classe AutoComparator2

```
import java.util.*;
public class AutoComparator2 implements Comparator<Auto>
{
   public int compare(Auto first, Auto second)
   {
      int compare1 = first.getChilometraggio()-second.getChilometraggio();
      if(compare1 != 0)
        return compare1;
      compare1 = first.getPrezzo() - second.getPrezzo();
      if(compare1 != 0)
        return compare1;
      compare1 = first.getNome().compareTo(second.getNome());
      if(compare1 != 0)
        return compare1;
     return first.getColore().compareTo(second.getColore());
   }
}
```

48

Cosa cambia solo nella stampa!!

```
private static void stampaAuto(List<Auto> l){
   System.out.println("In che ordine vuoi i risultati?");
   System.out.println("1) Prezzo > Chilometraggio > Colore > Nome");
   System.out.println("2) Chilometraggio > Prezzo > Nome > Colore");
   System.out.print("Inserisci la tua scelta: ");
   int valore = 1;
   try
   {
        valore = Integer.parseInt(lettore.leggiLinea());
   }
   catch(Exception e){
        System.out.println("Valore errato, si usa l'opzione di default");}
   switch(valore){
   case 1: break;
   case 2: Collections.sort(l, new AutoComparator2());   break;
   default: System.out.println("Errore, si usa l'opzione 1");
            break;}…//E dopo é come prima!!!!
```

49