Programmazione 2
Esercitazione 7

Livio Pompianu - Simone Porcu

Enum

I tipi enumerativi ci permettono di dichiarare delle variabili di tipo *enum* e possono assumere solo i valori elencati nella definizione del tipo enumerativo

```
public enum Settimana{
    LUNEDI,MARTEDI,MERCOLEDI,GIOVEDI,VENERDI,SABATO,DOMENICA;

}

import Settimana

public class Agenda{
    Settimana giorno = Settimana.LUNEDÌ;

    if(giorno == LUNEDÌ) {

        System.out.println(“datti malato”);

    }

}
```

Generics 1/6

Un obiettivo di primaria importanza nella programmazione è garantire il riuso del codice.

La soluzione si raggiunge spesso tramite tecniche di programmazione legate da un’unica idea: mantenere un elevato livello di astrazione.

Si parla di programmazione generica.

Generics 2/6

La programmazione generica può essere realizzata in Java tramite:

- ereditarietà;
- tipi parametrici.

Utilizzare tipi parametrici significa scrivere del codice che, applicato a tipi differenti, mantiene lo stesso comportamento.

Generics 3/6

Esempio, lista concatenata per interi e per stringhe

```
public class LinkedListString {
     public LinkedListString() { … }
     public boolean add(String e){ … }
     public String get(int index) { … }
}

public class LinkedListInteger {
     public LinkedListInteger() { … }
     public boolean add(Integer e){ … }
     public Integer get(int index) { … }

}
```

Generics 3/6

Esempio, lista concatenata per interi e per stringhe

```
public class LinkedListString {
    public LinkedListString() { … }
    public boolean add(String e){ … }
    public String get(int index) { … }
}

public class LinkedListInteger {
    public LinkedListInteger() { … }
    public boolean add(Integer e){ … }
    public Integer get(int index) { … }

}
```

NON DUPLICARE IL CODICE

Generics 4/6

Utilizzando i parametri di tipo posso scrivere classi senza conoscere effettivamente il tipo utilizzato

Le Collection utilizzano tutte i tipi parametrici

```
public class LinkedList<E> {
    public LinkedList() { … }

    public boolean add(E e){ … }

    public E get(int index) { … }

}
```

Generics 5/6

La classe precedente può essere utilizzata nel proprio codice nel seguente modo:

```
LinkedList<String> lista1 = new LinkedList<>();

LinkedList<Integer> lista2 = new LinkedList<>();


lista1.add("ciao");          // Compila


lista2.add(5);               // Compila

lista2.add(true);            // Errore in compilazione
```

Generics 6/6

Può essere necessario restringere il tipo passato come argomento

Esempio: una classe che lavora solo con tipi numerici

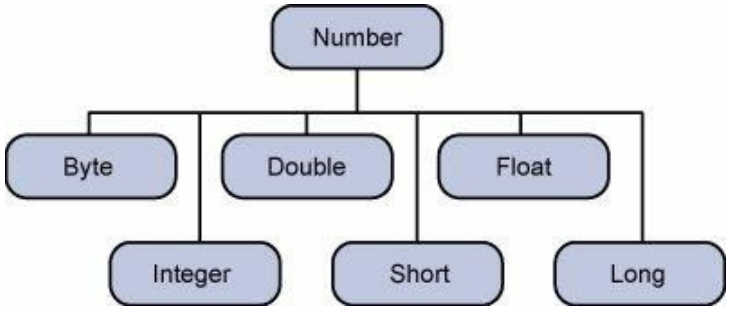
```
public class Point<T extends Number> {

    private T x;
    private T y;

    public T getX() { return this.x; }

 }

Point<Double> p = new Point<>();
```



Nel caso si debba specificare un'interfaccia si usa la keyword **`implements`**

Comparable 1/3

Il metodo equals permette di stabilire se due oggetti sono uguali secondo un criterio fissato: descrive una relazione di equivalenza.

In svariati contesti, è utile definire inoltre una relazione d’ordine tra elementi di una stessa classe.

Senza tale relazione non è possibile applicare un algoritmo di ordinamento.

Comparable 2/3

L’interfaccia **`comparable`** fornisce un unico metodo, **`compareTo`**, esempio: `a.compareTo(b)`.

L’obiettivo del metodo, considerato l’oggetto attuale e l’oggetto in input, è determinare qual è l’ordine tra di essi (maggiore a, maggiore b, uguali).

Se l’oggetto attuale **a** è maggiore rende un valore positivo, zero se sono uguali, negativo quando il maggiore è **b**.

Comparable 3/3

Quando si definisce il metodo compareTo, si sta definendo una relazione d'ordine.

Una relazione d'ordine deve essere:

- riflessiva;
- antisimmetrica;
- transitiva.

Ora è possibile ordinare elementi, ad esempio un array si ordina con **`Arrays.sort(mioArray)`**.

Esempio Comparable

```
public final class Integer implements Comparable<Integer> {

    public int compareTo(Integer x) {
        return (this < x) ? -1 : ((this == x) ? 0 : 1);
    }
}

Integer n = new Integer(5);
Integer m = new Integer(10);

assert n.compareTo(n) == 0;
assert n.compareTo(5) == 0;       // boxing

assert n.compareTo(m) == -1;
assert m.compareTo(n) == 1;
```

Iterable

Un oggetto che implementa l'interfaccia Iterable<T> può essere usato nei cicli for-each

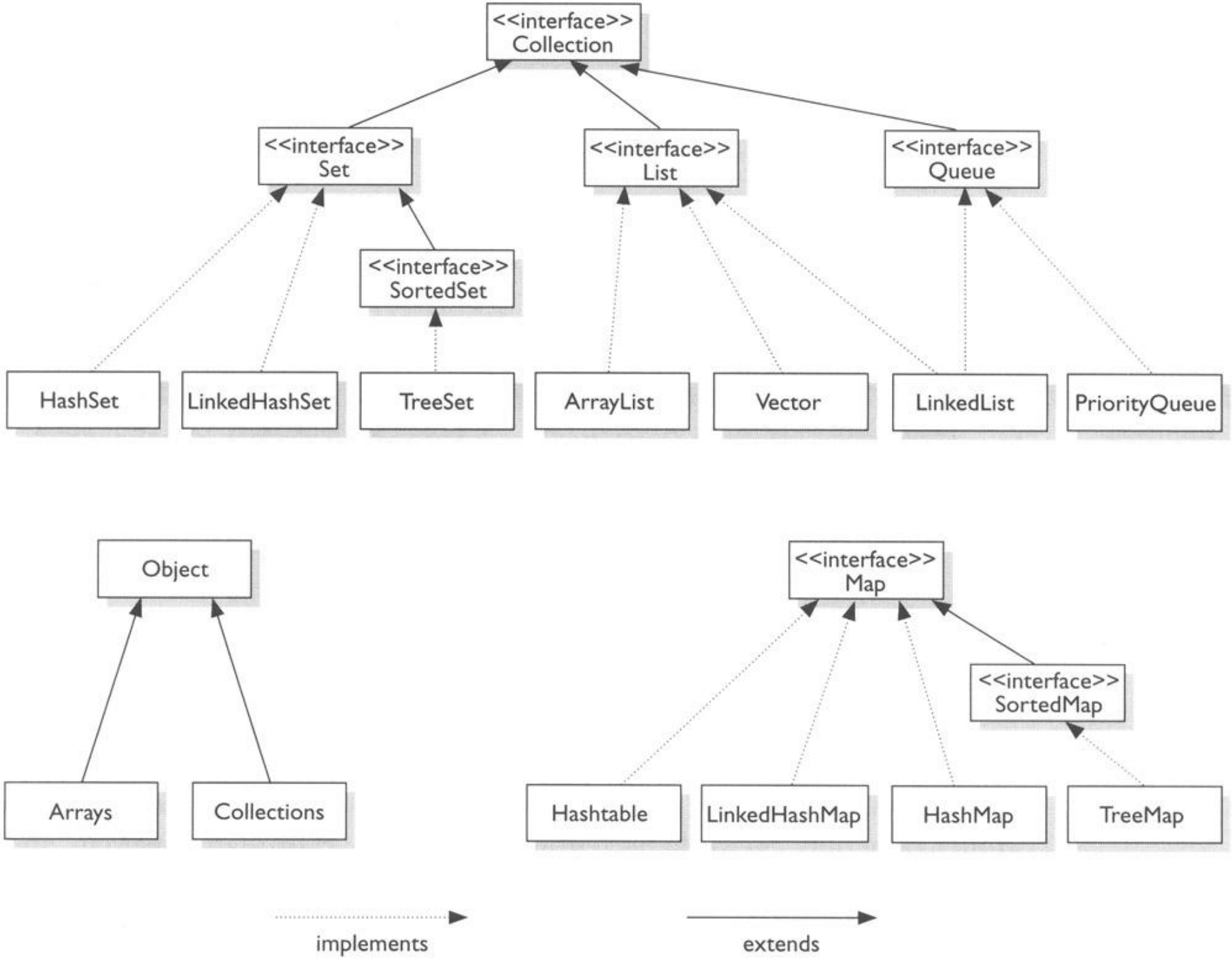
```
MyList<String> list = ...;
for(String s : list) { ... }
```

Per implementare Iterable è necessario il metodo Iterator<T> iterator()

```
public class MyList<T> implements Iterable<T> {
    MyListIterator<T> iterator(){ ... }
}
```

`MyListIterator` deve a sua volta implementare l'interfaccia `Iterator<T>`, ovvero i metodi `hashNext()` e `next()`

`Array` e implementazioni di `Collection` implementano `Iterable`


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
<<interface>>
SortedSet
HashSet
LinkedHashSet
TreeSet
ArrayList
Vector
LinkedList
PriorityQueue
Object
Arrays
Collections
<<interface>>
Map
<<interface>>
SortedMap
Hashtable
LinkedHashMap
HashMap
TreeMap
implements
extends

Map<K, V> 1/2

Le mappe sono associazioni tra chiavi (K) e valori (V)

Esempio: contiamo le occorrenze delle lettere dell'alfabeto

```
Map<String, Integer> mappa = new HashMap<>();
mappa.put("a", 1);
```

Key: String	Value: Integer
"a"	new Integer(1)

```
assert mappa.get("a") == 1; //abbiamo associato ad "a" il
//valore 1
assert mappa.get("b") == null; //non abbiamo associato
//niente a b, la mappa restituisce null
```

Map<K, V> 2/2

Per aggiornare una entry, dobbiamo prima verificare se il valore è presente

```
if(mappa.contaisKey(“a”)) { //se c’è devo incrementare il valore
    Integer numA = mappa.get(“a”);
    numA++;
    mappa.put(“a”, numA);

else { //se non c’è aggiungo la entry
    mappa.put(“a”, 1);
}
```

Nota: potevo controllare se `mappa.get(“a”) == null` anziché usare `containsKey()`

Esercizi

Esercizio GenericsHashMap 1/3

Creare una classe **GenericsHashMap** per gestire coppie di elementi generici: sia la chiave che il valore possono avere tipi arbitrari.

La coppia di elementi deve essere espressa attraverso la classe **Pair**. La struttura data quindi è un insieme non ordinato di coppie di elementi generici, che si traduce in un **HashSet** di Pair.

Esercizio GenericsHashMap 2/3

L'unico attributo di GenericsHashMap è l'HashSet di coppie. I metodi sono i seguenti:

- **put**
- **get**
- **remove**
- **size**
- **getKeys**
- **getValues**

Esercizio GenericsHashMap 3/3

La classe Pair contiene come unici attributi i due elementi della coppia. Vi sono i metodi:

- **getFirst**
- **getSecond**
- **equals**
- **hashCode**

Si utilizzi la classe di **MapTester** fornita.