Programmazione Orientata agli Oggetti

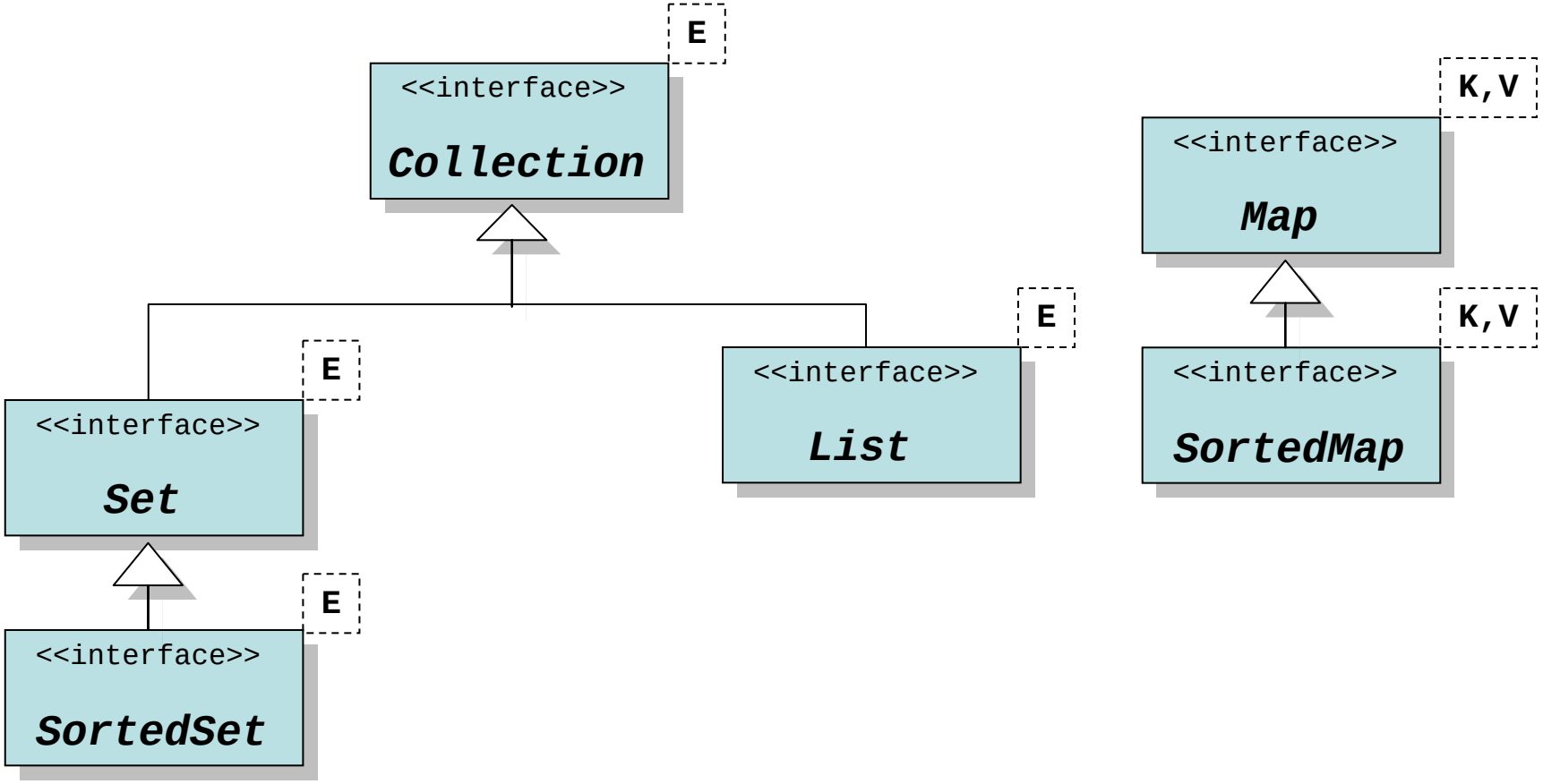
Collezioni

Liste generiche

Concetti introdotti

- Liste Generiche
- Iteratori Generici
- Boxing-unboxing
- Ordinamento **`Comparable`**, **`Comparator`**

Collezioni: Interface



La interface: `Collection<E>`

- L'interface **`Collection<E>`** dichiara i metodi di una collezione generica
- Questi metodi permettono di svolgere operazioni quali:
  - Aggiungere un elemento alla collezione
  - Verificare la dimensione della collezione
  - Verificare se la collezione è vuota
  - Aggiungere tutti gli elementi di un'altra collezione
  - Ottenere un *iteratore* con cui scandire la collezione

La interface: Collection<E>

```
public interface Collection<E> extends Iterable<E> {
   //Basic operations
        int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(E element); //optional
        boolean remove(Object element); //optional
        Iterator<E> iterator();
   //Bulk operations
        boolean containsAll(Collection<?> c);
        boolean addAll(Collection<? extends E> c); //optional
        boolean removeAll(Collection<?> c); //optional
        boolean retainAll(Collection<?> c); //optional
        void clear(); //optional
   //Array operations
        Object[] toArray();
        <T> T[] toArray(T[] a);
}
```

Collection<E>: metodi base

- **`int size();`**
  ritorna il numero di elementi presenti nella collezione

- **`boolean isEmpty();`**
  ritorna **`true`** se la collezione è vuota

- **`boolean contains(Object element);`**
  ritorna **`true`** se la collezione contiene un elemento uguale a quello passato come parametro (l'uguaglianza è verificata dal metodo **`equals()`**)

- **`boolean add(E element);`**
  aggiunge alla collezione l'elemento passato; ritorna **`true`** se la collezione è cambiata dopo la chiamata a questo metodo

- **`boolean remove(Object element);`**
  rimuove dalla collezione gli elementi uguali all'oggetto passato come parametro (l'uguaglianza è verificata dal metodo **`equals()`**). Ritorna **`true`** se la collezione è cambiata dopo l'invocazione del metodo

- **`Iterator<E> iterator();`**
  restituisce un oggetto Iterator, per iterare sugli elementi della collezione

Collection<E>: metodi bulk

- **`boolean containsAll(Collection<?> c);`**
  ritorna **`true`** se la collezione contiene tutti gli elementi della collezione passata come parametro
- **`boolean addAll(Collection<? extends E> c);`**
  aggiunge alla collezione tutti gli elementi della collezione passata come parametro; ritorna **`true`** se la collezione è cambiata dopo l'invocazione di questo metodo
- **`boolean removeAll(Collection<?> c);`**
  rimuove dalla collezione tutti gli elementi uguali (l'uguaglianza è verificata dal metodo **`equals()`**) che sono contenuti nella collezione passata come parametro; ritorna **`true`** se la collezione è cambiata dopo l'invocazione di questo metodo
- **`boolean retainAll(Collection<?> c);`**
  rimuove dalla collezione tutti gli elementi che non sono presenti nella collezione passata come parametro; ritorna **`true`** se la collezione è cambiata dopo l'invocazione di questo metodo
- **`void clear();`**
  rimuove tutti gli elementi dalla collezione

Iterazione: interface `Iterator<E>`

- L'iterazione di una collezione avviene attraverso un oggetto che ha la responsabilità di governare l'iterazione
- Questo oggetto, che viene chiesto alla collezione mediante il metodo **`iterator()`**, implementa l'interface **`Iterator<E>`**, che offre i metodi
  - **`boolean hasNext()`**
  - **`E next()`**
  - **`void remove()`**

Iterator<E>: metodi

- **`boolean hasNext();`**

  ritorna **`true`** se e solo se esiste un altro elemento da scandire

- **`E next();`**

  restituisce il prossimo elemento della collezione nella scansione corrente

- **`void remove();`**

  rimuove dalla collezione l'ultimo elemento che è stato restituito dalla chiamata di **`next()`**

`Iterator<E>`: iterazione

- La chiamata ripetuta di **`next()`** permette di scorrere gli elementi della collezione uno alla volta
- Se si raggiunge la fine della collezione viene sollevata una eccezione (che interrompe il programma) **`java.util.NoSuchElementException`**
- Per evitare questa situazione, prima di chiamare **`next()`** si usa il metodo **`hasNext()`**, che ritorna **`true`** se e solo se esiste un altro elemento su cui iterare

Iterator<E>: rimozione elementi

- Il metodo **`remove()`** rimuove l'elemento restituito dall'ultima chiamata di **`next()`**
- Non è ammesso chiamare **`remove()`** se prima non si è chiamato **`next()`**

- Es. voglio eliminare due elementi consecutivi:

```
it.remove();
it.remove();   // ERRORE
```

devo prima chiamare **`next()`**:

```
it.remove();
it.next();
it.remove();  // OK
```

Iterazione: for-each

- Per iterare su **tutti** gli elementi di una collezione è possibile usare la forma "for-each" dell'istruzione **`for`**

  `for(`*Tipo elemento* **`:`** *collezione***`)`**

  istruzione_su elemento

Collezioni e tipi primitivi

- Nelle collezioni non si possono memorizzare tipi primitivi
- Se si vogliono gestire tipi primitivi è necessario usare le classi wrapper
- Esempio: una collezione di interi

```
Collection<Integer> c;
```

- Dalla versione 1.5 di Java, la gestione di oggetti wrapper è semplificata dalle funzionalità di *boxing* ed *unboxing*

Boxing

- *Boxing*: è possibile assegnare direttamente tipi primitivi a oggetti wrapper
- Le seguenti istruzioni sono equivalenti:

```
int i = 0;
Integer iWrap;
iWrap = i;
iWrap = 5;
```

```
int i = 0;
Integer iWrap;
iWrap = new Integer(i);
iWrap = new Integer(5);
```

- È il compilatore che inserisce le istruzioni per gestire il wrapping

Unboxing

- *Unboxing*: è possibile assegnare direttamente oggetti wrapper a tipi primitivi
- Le seguenti istruzioni sono equivalenti:

```
int i = 0;
Integer iWrap;
iWrap = 5;
i = iWrap;
```

```
int i = 0;
Integer iWrap;
iWrap = new Integer(5);
i = iWrap.intValue();
```

- È il compilatore che inserisce le chiamate a costruttori e metodi

Boxing, unboxing e collezioni

- Grazie a boxing e unboxing, anche la gestione delle collezioni che memorizzano informazioni riconducibili a tipi primitivi è semplificata
- Le seguenti operazioni sono lecite (grazie a boxing e unboxing):

```
Collection<Integer> c;
c = new LinkedList<Integer>();
int i = 4;
c.add(i);
c.add(5);
```

Attenzione allo zucchero sintattico

- Le nuove versioni del compilatore tendono a semplificare la gestione dei tipi primitivi
- Tuttavia, è necessario comprendere a fondo
  - la differenza tra il concetto di tipo primitivo e la loro controparte ad oggetti, i wrapper
  - quali operazioni non sono necessarie solo grazie ai servizi offerti dalle ultime versioni del compilatore java (sarebbero necessarie con versioni precedenti)
  - quali operazioni il compilatore inserisce per conto nostro
- Perché conviene avere queste competenze?
  - per stimare meglio il numero di oggetti creati dalle nostre applicazioni
  - per migliorare la nostra capacità di ricerca dell'origine degli errori sia a tempo di compilazione che di esecuzione
  - per riuscire ad usare versioni precedenti del compilatore

Liste: interface `List<E>`

- Una lista è una collezione che mantiene gli elementi ordinati secondo l'ordine di inserimento (il primo elemento aggiunto alla lista è in prima posizione, il secondo in seconda posizione, …, l'ultimo elemento aggiunto è in ultima posizione)
- L'interface **`List<E>`** estende l'interface **`Collection<E>`**
  - Oltre ai metodi della interface **`Collection<E>`**, **`List<E>`** offre metodi che consentono accesso e inserimento indicizzati degli elementi

Implementazioni di `List`

- Il package **`java.util.*`** offre due diverse implementazioni di **`List`**
  - **`ArrayList`**
  - **`LinkedList`**

Esempio

```
public class ListTest {
  @Test
   public void testRemoveAll() {
       Collection<Integer> c = new LinkedList<Integer>();
       Collection<Integer> t = new LinkedList<Integer>();

       c.add(1);
       c.add(2);
       c.add(3);
       t.add(1);
       t.add(2);
       assertTrue(c.removeAll(t));
       Iterator<Integer> it = c.iterator();
       assertTrue(it.hasNext());
       assertEquals(3,it.next());
       assertFalse(it.hasNext());
   }
}
```

Esempio

- Codice discusso a lezione

Funzioni di utilità su liste

- La classe **`java.util.Collections`**
  - offre un vasto insieme di metodi (statici) generici che implementano utili algoritmi per la manipolazione di liste
  - vedi documentazione

Ordinamenti e ricerche

- Abbiamo metodi che implementano algoritmi efficienti per
  - ordinare una lista
  - ricercare la posizione di un elemento in una lista ordinata
  - ricercare l'elemento "più grande"/"più piccolo" in una lista

Ordinamenti e ricerche

- Queste operazioni hanno un senso se esiste una relazione d'ordine tra gli elementi della lista
  - in altri termini, gli elementi della lista devono sapersi confrontare
  - oppure ci deve essere un oggetto esterno che sa come confrontare due oggetti della lista

Definire un criterio di ordinamento

- I metodi (per le operazioni di ordinamento e ricerca) della classe **`Collections`** si affidano all'esistenza di un criterio di ordinamento specifico per il tipo degli elementi contenuti nella collezione (o suo supertipo)
- La responsabilità di modellare il criterio di ordinamento può essere affidata, in alternativa:
  - alla stessa classe degli oggetti contenuti, che deve implementare una apposita interfaccia **`java.lang.Comparable`**
  - ad una classe esterna alla classe degli oggetti contenuti; tale classe esiste solo con l'obiettivo di confrontarli, si chiama *comparatore* e rispetta l'interfaccia **`java.util.Comparator`**

L'interface `java.lang.Comparable<T>`

- L'interface **`java.lang.Comparable<T>`** ha un solo metodo:

```
public int compareTo(T that)
```

che deve restituire un valore che è:

- minore, uguale, maggiore di zero a seconda che l'oggetto corrente sia
- minore, uguale, maggiore dell'oggetto riferito dal parametro **`that`**

L'interface `java.lang.Comparable<T>`

- Molte importanti classi della libreria standard implementano **`java.lang.Comparable<T>`**, es.
  - **`java.lang.String`**
  - **`java.util.Calendar`**
  - **`java.util.Date`**
  - **`java.io.File`**
  - **`java.net.URI`**
  - tutte le classi wrapper
  - … e molte altre ancora

L'interface java.lang.Comparable<T>: esempio

```
public class Persona implements Comparable<Persona> {
   private String nome;

   public Persona(String nome) {
       this.nome = nome;
   }

   public String getNome() {
       return this.nome;
   }

   public int compareTo(Persona p) {
       return this.nome.compareTo(p.getNome());
   }
}
```

L'interface java.lang.Comparable<T>: esempio

```
public class PersonaTest {
  @Test
  public void testCompareTo() {
    Persona p1 = new Persona("Paolo");
    Persona p2 = new Persona("Valter");

    assertTrue(p1.compareTo(p2) < 0);   // <0

    Persona p3 = new Persona("Paolo");
    assertTrue (p1.compareTo(p3) == 0); // 0

    Persona p4 = new Persona("Anna");
    assertTrue (p1.compareTo(p4) > 0);  // >0
  }
}
```

Ordinamento "naturale"

- Su un oggetto **`List<T>`** contenente oggetti che implementano l'interfaccia **`java.lang.Comparable<T>`**
  - si possono effettuare ricerche
  - si può calcolare il massimo e il minimo
  - si può effettuare l'ordinamento
- Queste operazioni si basano sull'ordinamento "naturale", ovvero sulla relazione d'ordine implementata dal metodo **`compareTo()`**

Ordinare una lista

- Una lista **`List<T>`** i cui elementi implementino l'interface **`Comparable<T>`** può essere ordinata (secondo l'ordinamento naturale) mediante il metodo statico **`Collections.sort()`**

  - NOTA: se gli elementi della lista **`List<T>`** non implementano l'interface **`java.lang.Comparable<T>`** si solleva un errore a tempo di compilazione

Ordinare una lista

```
public class SortTest {
  @Test
   public void testSort() {
     List<Persona> l = new LinkedList<Persona>();
     l.add(new Persona(“Valter”));
     l.add(new Persona(“Paolo”));
     l.add(new Persona(“Giacomo”));
     l.add(new Persona(“Alessandro”));
     Collections.sort(l);
     assertEquals(“Alessandro”,l.get(0).getNome());
     assertEquals(“Giacomo”,l.get(1).getNome());
     assertEquals(“Paolo”,l.get(2).getNome());
     assertEquals(“Valter”,l.get(3).getNome());
}
```

NOTA: se **`Persona`** non implementasse **`Comparable<Persona>`**, si solleverebbe un errore a tempo di compilazione

Ottenere l'elemento max/min di una lista

- Da una lista **`List<T>`** i cui elementi implementino l'interface **`java.lang.Comparable<T>`** può essere ottenuto l'elemento massimo/minimo (rispetto all'ordinamento naturale) mediante il metodo statico **`Collections.max()/Collections.min()`**
  - NOTA: se gli elementi della lista **`List<T>`** non implementano l'interface **`java.lang.Comparable<T>`** si solleva un errore a tempo di compilazione

Ottenere l'elemento max/min di una lista

```
public class SortTest {
  @Test
   public void testSort() {
     List<Persona> l = new LinkedList<Persona>();
     l.add(new Persona(“Valter”));
     l.add(new Persona(“Paolo”));
     l.add(new Persona(“Giacomo”));
     l.add(new Persona(“Alessandro”));

     assertEquals(“Alessandro”,Collections.min(l).getNome());
     assertEquals(“Valter”, Collections.max(l).getNome());
}
```

NOTA: se **Persona** non implementasse **Comparable<Persona>**, si solleverebbe un errore a tempo di compilazione

L'interface `java.util.Comparator<T>`

- Se vogliamo ordinare una lista secondo un criterio diverso dall'ordinamento naturale?
- La classe **`Collections`** offre una versione del metodo *sort()* che si affida ad un oggetto esterno, passato come parametro, che sa effettuare i confronti necessari all'ordinamento

```
Collections.sort(
        List<T> listaDaOrdinare,
        Comparator<? super T> comparatore
      )
```

L'interface `java.util.Comparator<T>`

- L'interfaccia **`java.util.Comparator<T>`** ha un metodo:

```
public int compare(T o1, T o2)
```

  che deve restituire un valore che è
  - minore, uguale, maggiore di zero a seconda che l'oggetto riferito da **`o1`** sia
  - minore, uguale, maggiore dell'oggetto riferito dal parametro **`o2`**
- N.B. è simile ma non identico al metodo **`compareTo()`** di **`Comparable<T>`**

Ordinare una lista con `java.util.Comparator<T>`

- Ordinamenti e ricerche possono essere effettuate anche facendo affidamento ad oggetti istanza di classi che implementano **`Comparator<T>`**

Ordinamenti

- In sostanza, se abbiamo bisogno di operare ordinamenti (o altre operazioni basate su una relazione d'ordine) su una lista **`List<T>`**
  - Gli elementi della lista devono implementare l'interface **`java.lang.Comparable<T>:`** la relazione d'ordine rappresentata da questa implementazione corrisponde all'ordinamento naturale degli elementi
- Se abbiamo bisogno di effettuare ordinamenti su relazioni d'ordine diverse da quella naturale, allora possiamo definire un'implementazione di **`java.util.Comparator<T>`**

Nota

- L'interface **`Comparable<T>`** è nel package **`java.lang`**, quindi non è necessario importarla
- L'interface **`java.util.Comparator<T>`** è nel package **`java.util`**, quindi va importata