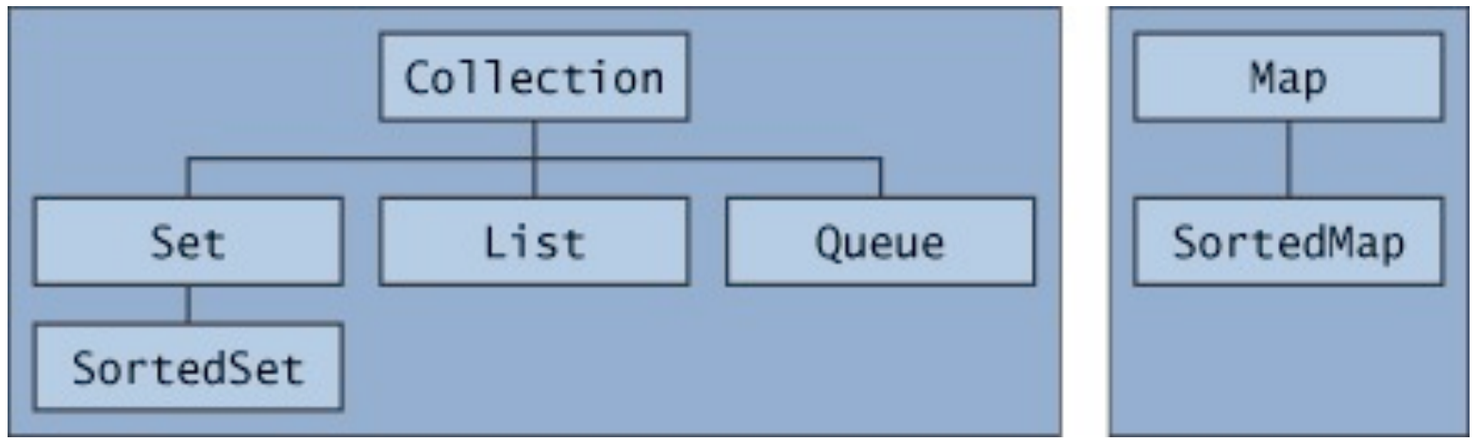
Collections

Collections

Oggetti della libreria standard di java
che possono contenere altri oggetti



Collection

Interfaccia molto generale

Permette di aggiungere/rimuovere elementi e di verificare se un oggetto è contenuto nella collezione

E' implementata da diverse classi,
ciascuna con caratteristiche diverse

`Set`

Rappresenta il concetto matematico di insieme

Non può contenere elementi duplicati

Interfaccia implementata da:
`HashSet` (più veloce),
`TreeSet` (elementi ordinati `SortedSet`) e
`LinkedHashSet` (ordine di inserimento)

List

Aggiunge l'accesso posizionale

```
list.get(index);
list.set(index,element);
```

Supporto la ricerca di elementi

```
list.indexOf(element);
list.lastIndexOf(element);
```

Permette di effettuare operazioni su range

```
list.sublist(from, to);
```

`List`

Interfaccia implementata da:
`ArrayList` (accessi più veloci veloce),
`LinkedList` (migliore per inserimenti/rimozioni),
`Vector`

Come al solito, usando l'interfaccia possiamo cambiare l'implementazione in base alle nostre esigenze

Map

Permette di associare **chiavi** e **valori**

Non ammette chiavi duplicate

Accesso basato sulla chiave

```
map.put(key,element);
map.get(key);
map.remove(key);
```

Offre diversi tipi di “viste”

```
Set keys = map.keySet();
Collection values = map.values();
map.containsKey(key);
map.containsValue(value);
```

`Map`

Anche in questo caso abbiamo
diverse implementazioni:
`HashMap`
`TreeMap`
`LinkedHashMap`

Valgono le stesse considerazioni
fatte per `Set`

Note

Gli elementi di un `Set` e le chiavi di una `Map` utilizzano `hashCode` ed `equals`

Gli oggetti usati con queste classi li devono implementare correttamente

Attenzione ai tipi di dati mutabili!

Queue

Metodi *speciali* per inserimento/rimozione

Con eccezioni	Con valori speciali
`add(element)`	`offer(element)`
`remove()`	`poll()`
`element()`	`peek()`

La politica di gestione della coda dipende dall'implementazione:

`LinkedList` (FIFO)

`PriorityQueue` (Elementi ordinati)

Note

Quando serve un concetto di ordinamento
(Come in `SortedSet` e `PriorityQueue`)
dobbiamo fornire una relazione d'ordine

Se la classe ha un ordinamento naturale può estendere `Comparable`, altrimenti dobbiamo fornire un `Comparator`

Comparable

Le classi che la implementano hanno un ordine naturale

```
public class A implements Comparable<A>{

   public int compareTo(A other) {
      // -1 se this < other
      //  0 se this = other
      // +1 se this > other
   }

}
```

Comparator

Utile per aggiungere nuove (anche multiple) relazioni d'ordine

```
public class C implements Comparator<A>{

    public int compare(A arg0, A arg1) {
        // TODO Auto-generated method stub
    }

}
```

Algoritmi

La classe `Collections` mette a disposizione algoritmi per:

ordinamento
mescolamento
manipolazione dei dati
ricerca

Ordinamento

Funziona su oggetti di tipo `List`

```
Collections.sort(list);
```

E' possibile definire come effettuare l'ordinamento tramite un `Comparator`

```
Collections.sort(list,
comparator);
```

Mescolamento

Funziona su oggetti di tipo `List`

```
Collections.shuffle(list);
```

Ordina in modo casuale gli elementi della lista

Manipolazione dei dati

Funzionano su oggetti di tipo `List`

```
Collections.reverse(list);
```

```
Collections.fill(list,value);
```

```
Collections.copy(destination, source);
```

```
Collections.swap(list,index1,index2);
```

Ricerca

Funziona su oggetti di tipo `List` ordinati

```
Collections.binarySearch(list, key);
```

Anche in questo caso possiamo usarlo con dei comparatori