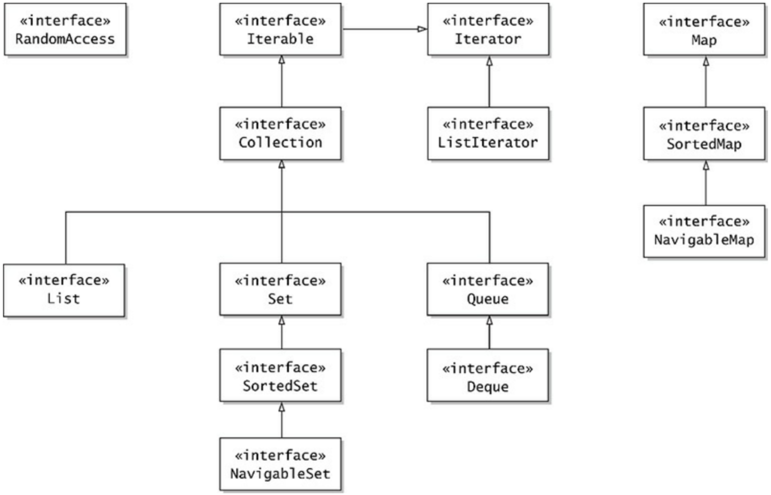
Collections

Prof. Michele Loreti

Programmazione Avanzata
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Java Collections. . .

Interfaces. . .



Interface Collection. . .

Interface `Collection<E>` defines a collection of elements having type `E`.

Interface Collection. . .

Interface Collection<E> defines a collection of elements having type E.

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)

void clear( )
boolean remove(Object o)
boolean remove(Collection<?> o)
boolean retainAll(Collection<?> o)
boolean removeIf(Predicate<? super E> filter)

int size( )
boolean isEmpty( )
boolean contains( Object o )
boolean containsAll(Collection<?> c)

Object[] toArray( )
T[] toArray(T[] a)
```

Interface List...

A `List` collection is an ordered collection where elements have position.

Interface List...

A List collection is an ordered collection where elements have position.

```
boolean add(int index, E e)
boolean addAll(int index, Collection<? extends E> c)

E get(int index)
E set(int index, E e)
E remove(int index)

int indexOf(Object o)
int lastIndexOf(Object o)

void replaceAll(UnaryOperator<E> operator)

void sort(Comparator<? super E> c)

static List<E> of(E ... elements)

List<E> subList(int fromIndex, int toIndex)
```

Interface List...

Classes implementing interface `List` provide methods to access the *n*th element of a list.

Interface List...

Classes implementing interface `List` provide methods to access the *n*th element of a list.

This operation may not be efficient!

Interface List...

Classes implementing interface `List` provide methods to access the *n*th element of a list.

This operation may not be efficient!

To indicate that operations `get( i )` are implemented efficiently, interface `RandomAccess` is implemented.

Interface List...

Classes implementing interface `List` provide methods to access the *n*th element of a list.

This operation may not be efficient!

To indicate that operations `get( i )` are implemented efficiently, interface `RandomAccess` is implemented.

Class `ArrayList` implements both `List` and `RandomAccess`. Class `LinkedList` only implements `List`.

Utility methods. . .

Class Collections provides utility methods that implement recurrent operations:

```
boolean disjoint(Collection<?> c1, Collection<?> c2)

boolean addAll(Collection<? super T> c, T ... elements)

void copy(List<? super T> dest, List<? extends T> src)

boolean replaceAll(List<T> list, T oldVal, T newVal)

void fill(List<? extends T> list, T obj)

List<T> nCopies(int n, T obj)

int frequency(Collection<?> c, Object o)
```

Iterator. . .

Each collection provides a way to iterate through its elements in some order.

Iterator. . .

Each collection provides a way to iterate through its elements in some order.

Interface `Collection<T>` extends interface `Iterable<T>` containing the method:

```
Iterator<T> iterator()
```

Iterator. . .

Each collection provides a way to iterate through its elements in some order.

Interface `Collection<T>` extends interface `Iterable<T>` containing the method:

```
Iterator<T> iterator()
```

Interface `Iterator<T>` provides the following methods:

```
boolean hasNext( )
```

```
T next( )
```

```
void remove( )
```

Iterator. . .

Each collection provides a way to iterate through its elements in some order.

Interface Collection <T> extends interface Iterable <T> containing the method:

```
Iterator<T> iterator()
```

Interface Iterator <T> provides the following methods:

```
boolean hasNext( )
```

```
T next( )
```

```
void remove( )
```

Warning: The default implementation of remove() throws an instance of UnsupportedOperationException and performs no other action.

Iterator. . .

Elements in a collection can be visited as follows:

```
Collection<String> coll = ...
...
Iterator<String> iterator = coll.iterator();
while (iterator.hasNext()) {
  String element = iterator.next();
  //Process element
  ...
}
```

Iterator. . .

Elements in a collection can be visited as follows:

```
Collection<String> coll = ...
...
Iterator<String> iterator = coll.iterator();
while (iterator.hasNext()) {
  String element = iterator.next();
  //Process element
  ...
}
```

Alternatively, we can use the *for-each* loop:

```
Collection<String> coll = ...
...
for (String element: coll) {
  //Process element
  ...
}
```

Iterator. . .

Removing elements. . .

The iterator of a collection can implement a specific remove method:

```
Collection<String> coll = ...
...
Iterator<String> iterator = coll.iterator();
while (iterator.hasNext()) {
  String element = iterator.next();
  if (exp) {
    iter.remove();
  }
}
```

Iterator. . .

Removing elements. . .

The iterator of a collection can implement a specific remove method:

```
Collection<String> coll = ...
...
Iterator<String> iterator = coll.iterator();
while (iterator.hasNext()) {
  String element = iterator.next();
  if (exp) {
    iter.remove();
  }
}
```

The easier way to remove elements from a collection is to use removeIf method:

```
coll.removeIf( element -> exp );//exp is the same as above!
```

List iterator. . .

From an instance of List<T> we can get a ListIterator <T>. This is an iterator for lists that allows us to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

List iterator. . .

From an instance of List<T> we can get a ListIterator <T>. This is an iterator for lists that allows us to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

```
 void add(T e)
 boolean hasNext( )
 boolean hasPrevious( )
T next( )
 int nextIndex( )
T previous( )
 int previousIndex( )
 void remove( )
 void set(T e)
```

Sets. . .

A Set<T> is a data structure where:

- we can efficiently compute if an element is in the set;
- the order of insertions is not remembered;
- elements are stored without multiplicity.

Sets. . .

A Set<T> is a data structure where:

- we can efficiently compute if an element is in the set;
- the order of insertions is not remembered;
- elements are stored without multiplicity.

Two implementations of interface Set<T> are:

- TreeSet<T>, set implemented via binary trees;
- HashSet<T>, set implemented via hash table.

Sets: HashSet. . .

In HashSet set is stored via a hash-table.

Sets: HashSet. . .

In HashSet set is stored via a hash-table.

Elements in the table are compared according to:

- int hashCode()
- boolean equals(Object o)

Sets: HashSet. . .

In HashSet set is stored via a hash-table.

Elements in the table are compared according to:

- `int hashCode()`
- `boolean equals(Object o)`

Warning: a correct implementation of these methods is crucial to avoid unexpected behaviours!

Sets...

Class TreeSet<T> should be used when we want to traverse the set in order:

- implements interfaces SortedSet<T> and NavigableSet<T>;
- either T implements Comparable<T> or a Comparator<T> must be provided.

Maps. . .

Maps store associations between keys and values (Map<K,V>).

Maps. . .

Maps store associations between keys and values (`Map<K,V>`).

Method `put(K k, V v)` adds a new association, or change the value of an existing key.

Maps. . .

Maps store associations between keys and values (`Map<K,V>`).

Method `put(K k, V v)` adds a new association, or change the value of an existing key.

Method `get(Object k)` retrieves the value associated with key `k`. If this value does not exist, `null` is returned.

Maps. . .

Maps store associations between keys and values (`Map<K,V>`).

Method `put(K k, V v)` adds a new association, or change the value of an existing key.

Method `get(Object k)` retrieves the value associated with key `k`. If this value does not exist, `null` is returned.

The method `getOrDefaultValue(Object k, V v)` can be used to get a default value if no value is associated with `k` (to avoid `NullPointerException`).

Maps. . .

Given a Map<K,V> we can get a view of keys, values and entries:

```
Set<K> keySet( )
Set<Map.Entry<K,V>> entrySet( )
Collection<V> values( )
```

Maps. . .

Given a Map<K,V> we can get a view of keys, values and entries:

```
Set<K> keySet( )
Set<Map.Entry<K,V>> entrySet( )
Collection<V> values( )
```

The collections returned by the method above are not copies! If you change it, you change the underlying map!

Maps. . .

Given a Map<K,V> we can get a view of keys, values and entries:

```
Set<K> keySet( )
Set<Map.Entry<K,V>> entrySet( )
Collection<V> values( )
```

The collections returned by the method above are not copies! If you change it, you change the underlying map!

An instance of Map.Entry<K,V> stores the entry in a map.

Maps. . .

Given a Map<K,V> we can get a view of keys, values and entries:

```
Set<K> keySet( )
Set<Map.Entry<K,V>> entrySet( )
Collection<V> values( )
```

The collections returned by the method above are not copies! If you change it, you change the underlying map!

An instance of Map.Entry<K,V> stores the entry in a map.

To iterate through the entries of a Map<K,V> we can also use the forEach method:

```
void forEach(BiConsumer<? super K,? super V> action)
```

Other collections. . .

- Properties: is used to store a persistent list of properties (like application specific settings).
- BitSet: implements a vector of bits that grows as needed.
- Stack: represents a last-in-first-out (LIFO) stack of objects.
- Queue: this is an interface that represents a collection designed for holding elements prior to processing.
- Deque: this is an interface describing a linear collection that supports element insertion and removal at both ends.
- PriorityQueue: An unbounded priority queue based on a priority heap.
- WeakHashMap: Hash table based implementation of the Map interface, with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use.

Views. . .

A collection view is a lightweight object that implements a collection interface, but doesn't store elements (examples are keySet and values methods of a map).

Views. . .

A collection view is a lightweight object that implements a collection interface, but doesn't store elements (examples are keySet and values methods of a map).

Examples:

```
 List.of( v1,...,vn )
Set.of( v1,...,vn )
```

To be continued...