Collections

Collections

Collections in java is a framework that provides an architecture to store and manipulate the group of *objects*. All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc, can be performed by java **Collections**.

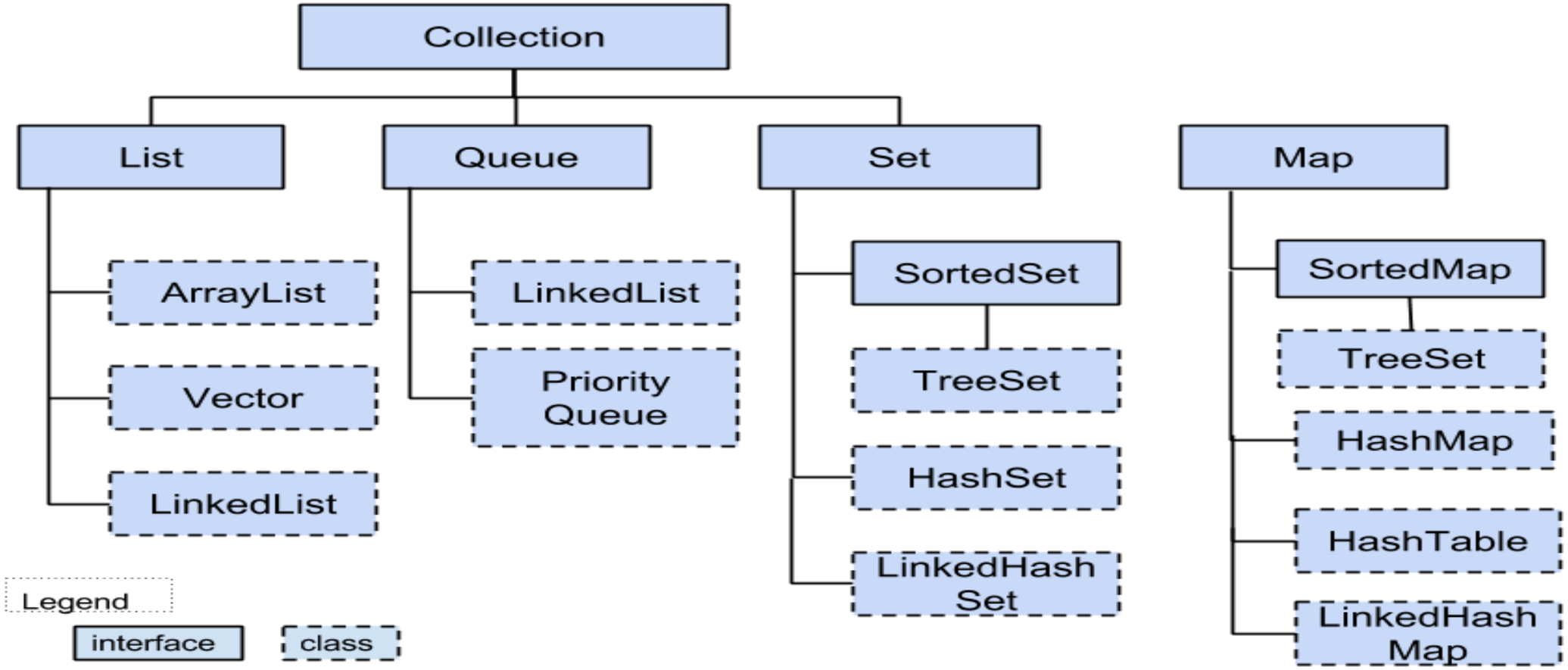
Java **Collection** simply means a single unit of *objects*. Java **Collection** framework provides many *interfaces* (*Set*, *List*, *Queue*, *Deque* etc.) and *classes* (*ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet* etc).

What is framework in java?

In short, framework provides premade architecture.
It represents a set of classes and interfaces

Hierarchy of Collection Framework

The *java.util* package contains all the *classes* and *interfaces* for **Collection** framework.



Methods of Collection interface

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public boolean addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no of element from the collection.

Methods of Collection interface

No.	Method	Description
8	public boolean contains(Object element)	is used to search an element.
9	public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
10	public Iterator iterator()	returns an iterator.
11	public Object[] toArray()	converts collection into array.
12	public boolean isEmpty()	checks if collection is empty.
13	Public boolean equals(Object element)	matches two collections.
14	public int hashCode()	returns the hashcode number for collection.

Iterator interface

It is one of the predefined interfaces present in **java.util.*** package. The purpose of this interface is that to extract or retrieve the elements of collection variable only in forward direction but not in backward direction. By default an object of **iterator** is pointing just before the first element of any collection framework variable.

Methods of Iterator interface

There are only three methods in the **Iterator** interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if iterator has more elements.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is rarely used.

Java List Interface

The **List** interface extends Collection and declares the behavior of a collection that stores a sequence of elements.

- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- In addition to the methods defined by Collection, List defines some of its own, which are summarized in the table in next slide.
- Several of the list methods will throw an *UnsupportedOperationException* if the collection cannot be modified, and a *ClassCastException* is generated when one object is incompatible with another.

Declared methods by List interface

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(int index, Collection c)	It is used to insert all of the elements in the specified collection into this list, starting at the specified position.
void get(int index)	Returns the object stored at the specified index within the invoking collection.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
ListIterator listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator listIterator(int index)	Returns an iterator to the invoking list that begins at the specified index.
Object remove(int index)	Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
Object set(int index, Object obj)	Assigns obj to the location specified by index within the invoking list.
List subList(int start, int end)	Returns a list that includes elements from start to end.1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Java ArrayList class

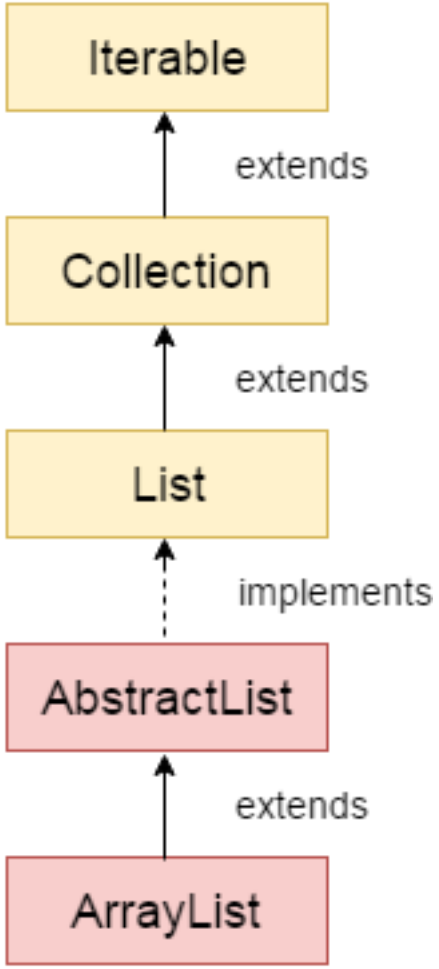
Java **ArrayList** class uses a dynamic array for storing the elements. It inherits **AbstractList** class and implements **List** interface.

The important points about Java **ArrayList** class are:

- Java **ArrayList** can contain duplicate elements.
- Java **ArrayList** maintains insertion order.
- Java **ArrayList** is non synchronized.
- Java **ArrayList** allows random access because array works at the index basis.
- In Java **ArrayList**, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

Hierarchy of ArrayList class

As shown in diagram to the right, Java **ArrayList** class extends **AbstractList** class which implements **List** interface. The **List** interface extends **Collection** and **Iterable** interfaces in hierarchical order.



Methods of Java ArrayList

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(Collection c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
void clear()	It is used to remove all of the elements from this list.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
Object[] toArray(Object[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean addAll(int index, Collection c)	It is used to insert all of the elements in the specified collection into this list, starting at the specified position.
Object clone()	It is used to return a shallow copy of an ArrayList.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Java ArrayList class - Example

```java
public static void main(String args[]) {
    // creating arraylist
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(10);
    numbers.add(20);
    numbers.add(30);
    // getting Iterator from arraylist to traverse elements
    Iterator itr = numbers.iterator();
    while (itr.hasNext()) {
        System.out.println(itr.next());
    }
}
```

```
run:
10
20
30
BUILD SUCCESSFUL (total time: 0 seconds)
```

Java Set Interface

A **Set** is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.
The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

The methods declared by set are summarized in the table to the right

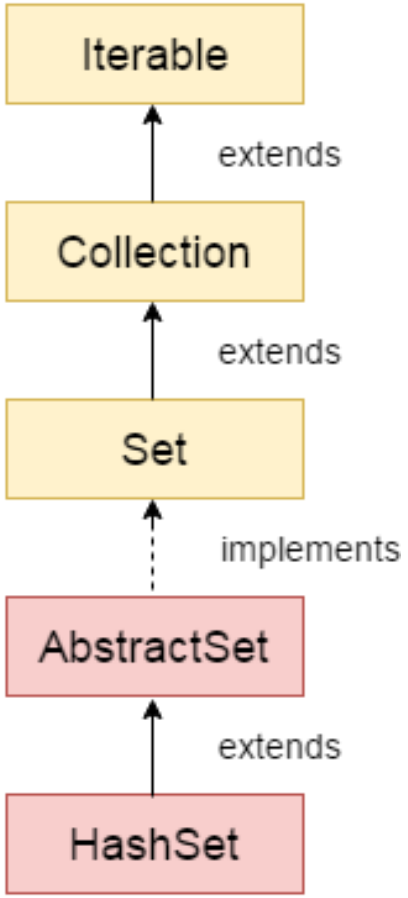
No.	Method & Description
1	**add( )** Adds an object to the collection.
2	**clear( )** Removes all objects from the collection.
3	**contains( )** Returns true if a specified object is an element within the collection.
4	**isEmpty( )** Returns true if the collection has no elements.
5	**iterator( )** Returns an Iterator object for the collection, which may be used to retrieve an object.
6	**remove( )** Removes a specified object from the collection.
7	**size( )** Returns the number of elements in the collection.

Java HashSet class

Java **HashSet** class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java **HashSet** class are:

- **HashSet** stores the elements by using a mechanism called *hashing*.
- **HashSet** contains unique elements only.



Hierarchy of HashSet class

The **HashSet** class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

Methods of Java HashSet class

Method	Description
void clear()	It is used to remove all of the elements from this set.
boolean contains(Object o)	It is used to return true if this set contains the specified element.
boolean add(Object o)	It is used to adds the specified element to this set if it is not already present.
boolean isEmpty()	It is used to return true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
Object clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
Iterator iterator()	It is used to return an iterator over the elements in this set.
int size()	It is used to return the number of elements in this set.

Java HashSet Example

```
public static void main(String args[]) {
    //Creating HashSet and adding elements
    HashSet<String> set = new HashSet<String>();
    set.add("Ali");
    set.add("Fredrik");
    set.add("Ulf");
    set.add("Eric");
    //Traversing elements
    Iterator<String> itr = set.iterator();
    while (itr.hasNext()) {
        System.out.println(itr.next());
    }
}
```

Result

```
run:
Fredrik
Eric
Ulf
Ali
BUILD SUCCESSFUL (total time: 0 seconds)
```

Could you figure out something here?

Java Map Interface

A **Map** contains values on the basis of key i.e. key and value pair. Each key and value pair is known as an entry. **Map** contains only unique keys.
Map is useful if you have to search, update or delete elements on the basis of key.

- Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using its key.
- Several methods throw a NoSuchElementException when no items exist in the invoking map.
- A ClassCastException is thrown when an object is incompatible with the elements in a map.
- A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.
- An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map.

Methods declared by Map interface

No.	Method & Description
1	**void clear( )** Removes all key/value pairs from the invoking map.
2	**boolean containsKey(Object k)** Returns true if the invoking map contains **k** as a key. Otherwise, returns false.
3	**boolean containsValue(Object v)** Returns true if the map contains **v** as a value. Otherwise, returns false.
4	**Set entrySet( )** Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.
5	**boolean equals(Object obj)** Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
6	**Object get(Object k)** Returns the value associated with the key **k**.
7	**int hashCode( )** Returns the hash code for the invoking map.

Methods declared by Map interface

No.	Method & Description
8	**boolean isEmpty( )** Returns true if the invoking map is empty. Otherwise, returns false.
9	**Set keySet( )** Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
10	**Object put(Object k, Object v)** Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
11	**void putAll(Map m)** Puts all the entries from **m** into this map.
12	**Object remove(Object k)** Removes the entry whose key equals **k**.
13	**int size( )** Returns the number of key/value pairs in the map.
14	**Collection values( )** Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Java HashMap class

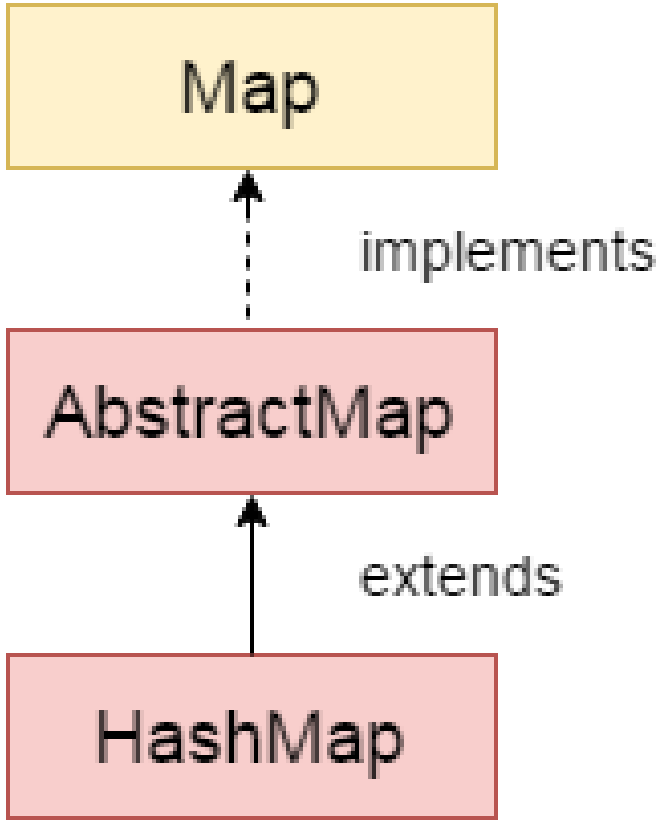
The **HashMap** *class* uses a *hashtable* to implement the Map *interface*. This allows the execution time of basic operations, such as *get*( ) and *put*( ), to remain constant even for large sets.

The important points about Java **HashMap** *class* are:

- A **HashMap** contains values based on the *key*.
- It contains only unique elements.
- It may have one *null key* and multiple *null* values.
- It maintains no order.

Hierarchy of HashMap class

As shown in the table to the right, **HashMap** class extends **AbstractMap** *class* and implements Map *interface*.



Methods of Java HashMap class

Method	Description
void clear()	It is used to remove all of the mappings from this map.
boolean containsKey(Object key)	It is used to return true if this map contains a mapping for the specified key.
boolean containsValue(Object value)	It is used to return true if this map maps one or more keys to the specified value.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
Object put(Object key, Object value)	It is used to associate the specified value with the specified key in this map.
int size()	It is used to return the number of key-value mappings in this map.
Collection values()	It is used to return a collection view of the values contained in this map.

Java HashMap - Traversing Example

```
public static void main(String args[]) {
    // create and populate hash map
    HashMap<String, String> hm = new HashMap<String, String>();
    hm.put("Ali@lexicon.se", "Ali");
    hm.put("Fredrik@lexicon.se", "Fredrik");
    hm.put("Ulf@lexicon.se", "Ulf");
    // traversing map
    for (Map.Entry m : hm.entrySet()) {
        System.out.println("Key: " + m.getKey() + ", Value: " + m.getValue());
    }
}
```

```
run:
Key: Fredrik@lexicon.se, Value: Fredrik
Key: Ali@lexicon.se, Value: Ali
Key: Ulf@lexicon.se, Value: Ulf
BUILD SUCCESSFUL (total time: 0 seconds)
```

Java HashMap – Single Value Example

```
public static void main(String args[]) {
    // create and populate hash map
    HashMap<String, String> hm = new HashMap<String, String>();
    hm.put("Ali@lexicon.se", "Ali");
    hm.put("Fredrik@lexicon.se", "Fredrik");
    hm.put("Ulf@lexicon.se", "Ulf");
    // getting specific value by key
    String value = hm.get("Ali@lexicon.se");
    // printing value
    System.out.println(value);
}
```

```
run:
Ali
BUILD SUCCESSFUL (total time: 0 seconds)
```

Sorting in Collection

Collections *class* provides static methods for sorting the elements of collection. If collection elements are of Set type, we can use TreeSet. But we cannot sort the elements of List. **Collections** *class* provides methods for sorting the elements of List type elements.

```
public static void main(String args[]) {
    // creating the list
    List<String> names = new ArrayList();
    names.add("Ulf");
    names.add("Fredrik");
    names.add("Ali");
    names.add("Kent");
    names.add("Eric");
    // sorting the list
    Collections.sort(names);
    // traversing the list
    Iterator itr = names.iterator();
    while (itr.hasNext()) {
        System.out.println(itr.next());
    }
}
```

```
run:
Ali
Eric
Fredrik
Kent
Ulf
BUILD SUCCESSFUL (total time: 0 seconds)
```

Java Comparable interface

Java **Comparable** *interface* is used to order the objects of user-defined *class*. This *interface* is found in *java.lang* package and contains only one method named *compareTo(Object).* It provide single sorting sequence only i.e. you can sort the elements on based on single data member only. For example it may be id, name, age or anything else.

```java
class Student implements Comparable<Student> {
    public int id;
    public String name;
    public int age;
    Student(int id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }
    public int compareTo(Student st) {
        if (age == st.age) {
            return 0;
        } else if (age > st.age) {
            return 1;
        } else {
            return -1;
        }
    }
}
```

```java
public static void main(String args[]) {
    // creating the list
    List<Student> students = new ArrayList();
    students.add(new Student(1, "Ulf", 35));
    students.add(new Student(2, "Fredrik", 39));
    students.add(new Student(3, "Ali", 28));
    students.add(new Student(4, "Eric", 29));
    // sorting the list
    Collections.sort(students);
    // traversing the list
    for(Student currentStudent : students)
        System.out.println("Id: " + currentStudent.id
                + ", Name: " + currentStudent.name
                + ", Age: " + currentStudent.age
        );
}
```

```
Id: 3, Name: Ali, Age: 28
Id: 4, Name: Eric, Age: 29
Id: 1, Name: Ulf, Age: 35
Id: 2, Name: Fredrik, Age: 39
BUILD SUCCESSFUL (total time: 0 seconds)
```

Questions?