Theory: The Collections Framework overview

3025 users solved this topic. Latest completion was about 3 hours ago.

21 minutes 0 / 5 problems solved Skip this topic Start practicing

Java provides the **collections framework** which consists of classes and interfaces for commonly reusable data structures such as lists, dynamic arrays, sets, and so on. The framework has a unified architecture for representing and manipulating collections, enabling collections to be used independently of implementation details via its interfaces.

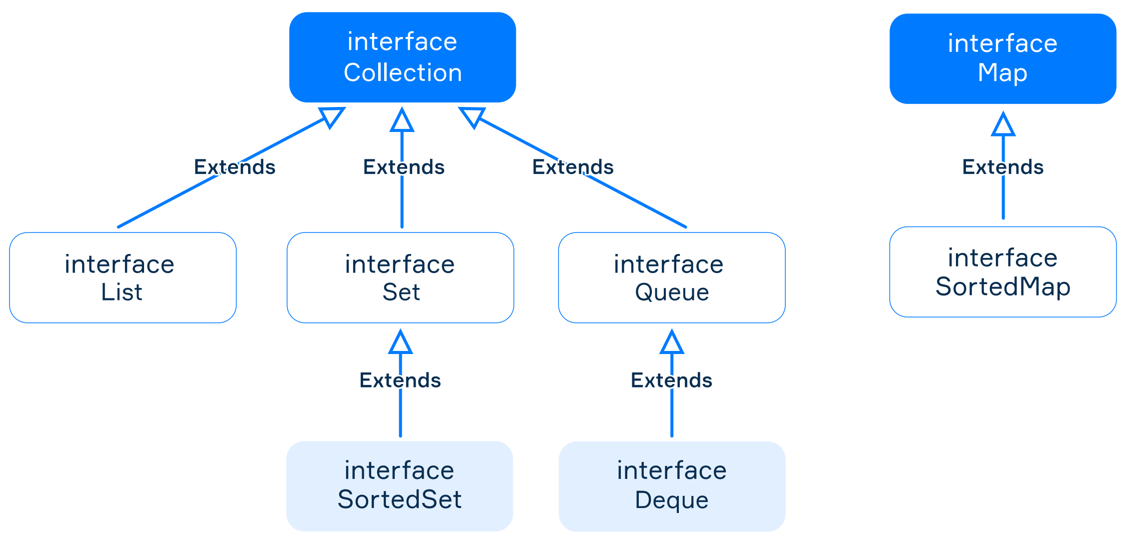
The framework includes:

- interfaces that represent different types of collections;
- primary implementations of the interfaces;
- legacy implementations from earlier releases (known as "old collections");
- special-purpose implementations (like immutable collections);
- algorithms represented by static methods that perform useful operations on collections.

In this topic, we will only consider the basic interfaces from the collections framework placed in the `java.util` package.

§1. Commonly used interfaces

There are two root generic interfaces `Collection<E>` and `Map<K,V>`, and some more specific interfaces to represent different types of collections.



The interface `Collection<E>` represents an abstract collection, which is a container for objects of the same type. It provides some common methods for all other types of collections.

The interfaces `List<E>`, `Set<E>`, `Queue<E>`, `SortedSet<E>`, and `Deque<E>` represent different types of collections. You cannot directly create an object of them since they are just interfaces. But each of them has several implementations. As an example, the `ArrayList` class, that represents a resizable array, is a primary representation of the `List<E>` interface. Other interfaces, as well as their implementations, will be considered in the following topics.

Another root interface is `Map<K,V>` that represents a map (or dictionary) for storing *key-value pairs* where `K` is the type of keys and `V` is the type of stored values. In the real world, a good example of a map is a phone book where keys are names of your friends and values are their phones. The `Map<K,V>` interface **is not** a subtype of the `Collection` interface, but maps are often considered as collections since they are part of the collection framework and have similar methods.

Note, the `Collection` and `Map` interfaces do not extend each other.

Current topic:

The Collections Framework overview

Topic depends on:

- × Interface
- × Generics and Object
- × What are collections

Topic is required for:

- List
- Set
- Queue and Stack

Table of contents:

§2. The Collection interface

Here are common methods provided by the `Collection` interface.

- `int size()` returns the number of elements in this collection;
- `boolean isEmpty()` returns `true` if this collection contains no elements;
- `boolean contains(Object o)` returns `true` if this collection contains the specified element;
- `boolean add(E e)` adds an element to the collection. Returns `true`, if the element was added, else returns `false`;
- `boolean remove(Object o)` removes a single instance of the specified element;
- `boolean removeAll(Collection<?> collection)` removes elements from this collection that are also contained in the specified collection;
- `void clear()` removes all elements from this collection.

It is possible to refer to any particular collection via this base interface since, as you know, the superclass can be used to refer to any subclass object derived from that superclass.

Let's create `languages` collection and add three elements to it:

```
Collection<String> languages = new ArrayList<>();

languages.add("English");
languages.add("Deutsch");
languages.add("Français");

System.out.println(languages.size()); // 3
```

This approach allows you to replace the concrete collection at any time without changing the code that uses it. It also fosters software reuse by providing a standard interface for collections and algorithms to manipulate them. It may sound complicated now, but the more you work with collections, the more understandable it will become.

It is impossible to get an element by index via the `Collection` interface because it is very abstract and does not provide such a method. But if it doesn't matter for you which particular collection to use, you can work via this interface.

It is important to understand that the order of the elements in the `ArrayList` is anyway preserved. We simply cannot call the `get` method via the `Collection` interface.

Every collection can be cast to a string by using `toString` and compared with another collection using the `equals` method. These methods come from `Object` and their behavior depends on elements stored in the collection and the type of the collection itself.

§3. Mutable and Immutable collections

All collections can be divided into two large groups: mutable and immutable. They both implement the `Collection<E>` interface, but immutable collections will throw `UnsupportedOperationException` when trying to invoke some methods which change them: for example, `add`, `remove`, `clear`.

In the next topics, we will consider how to create and when to use immutable collections. Now just remember that they are present here.

§4. Iterating over collections

If you would like to iterate over all elements of *any* collection, you can use the *for-each* style loop. Let's return to our `languages` collection:

```
for (String lang : languages) {
    System.out.println(lang);
}
```

This code prints all elements of this collection.

```
English
Deutsch
Français
```

The order of elements when iterating depends on a type of particular collection that is actually being used.

If you are already familiar with method references or lambda expressions, you can use another style for iterations using the `forEach(Consumer<T> consumer)` method:

```
languages.forEach(System.out::println); // with method reference
languages.forEach(elem -
> System.out.println(elem)); // with lambda expression
```

This looks very readable but is optional for use.

§5. Removing elements

It is also possible to remove elements from a mutable collection (like `ArrayList`).

```
languages.remove("Deutsch");

System.out.println(languages.size()); // 2
```

Note, the `remove` as well as the `contains` methods rely on the method `equals` of the elements. If you store non-standard classes in the collection, `equals` together with `hashCode` should be overridden.

Again, if you are already familiar with lambda expressions, you can invoke the `removeIf` method to remove all of the elements that satisfy the given predicate:

```
languages.removeIf(lang -> lang.startsWith("E")); // it removes English

System.out.println(languages.size()); // 1
```

Use any way you like.

§6. Conclusion

Java collections framework provides a set of interfaces with common methods for different types of collections. We've considered the `Collection<E>` interface which is an abstract container for storing values of the same type. Any particular collection (excluding maps) can be used through it in a program and can be iterated through by using the *for-each* loop or the `forEach` method.

For now, we confine ourselves to this much. All other interfaces including `Map<K,V>` will be considered in the following topics.

Report a typo

272 users liked this theory. 6 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (2) Hints (0) Useful links (1) Show discussion