Theory: List

17 minutes 0 / 5 problems solved

Skip this topic Start practicing

As you know, lists are the closest type to arrays, except their size can be changed dynamically while an array's size is constrained. Moreover, lists provide more advanced behavior than arrays. In this topic, you will deepen your knowledge of lists and their relationship with *the Collections Framework*.

> A list is an *ordered* collection of elements. It means that each element has a position in the list specified by an integer index like in regular arrays.

§1. The List interface

The `List<E>` interface represents a list as an abstract data type. It extends the `Collection<E>` interface acquiring its methods and adds some new methods:

- `E set(int index, E element)` replaces the element at the specified position in this list with the specified element and returns the element that was replaced;
- `E get(int index)` returns the element at the specified position in the list;
- `int indexOf(Object obj)` returns the index of the first occurrence of the element in the list or `-1` if there is no such index;
- `int lastIndexOf(Object obj)` returns the index of the last occurrence of the element in the list of `-1` if there is no such index;
- `List<E> subList(int fromIndex, int toIndex)` returns a sublist of this list from `fromIndex` included to `toIndex` excluded.

As you can see, the methods presume that a list is an ordered collection.

You cannot create an instance of the `List` interface, but you can create an instance of one of its implementations: `ArrayList` or `LinkedList` or an *immutable* list, and then use it through the common `List` interface. You will have access to all methods declared in both `List<E>` and `Collection<E>` interfaces.

> Working with lists through the `List` interface is considered good practice in programming since your code will not depend on the internal mechanisms of a specific implementation.

§2. Immutable lists

The simplest way to create a list is to invoke the `of` method of the `List` interface.

```
List<String> emptyList = List.of(); // 0 elements
List<String> names = List.of("Larry", "Kenny", "Sabrina"); // 3 elements
List<Integer> numbers = List.of(0, 1, 1, 2, 3, 5, 8, 13);  // 8 elements
```

It returns an **immutable** list containing either all the passed elements or an empty list. Using this method is convenient when creating list constants or testing some code.

Let's perform some operations:

Current topic:

List

Topic depends on

ArrayList

The Collections Framework overview Stage 6

Topic is required for

The utility class Collections

Iterator and Iterable

Functional data processing with streams

Callable and Future

Topological sort

Table of contents:

↑ Theory: List

§1. The List interface

§2. Immutable lists

§3. Mutable lists

§4. Iterating over a list

§5. List equality

Feedback & Comments

```
List<String> daysOfWeek = List.of(
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
);



System.out.println(daysOfWeek.size()); // 7

System.out.println(daysOfWeek.get(1)); // Tuesday

System.out.println(daysOfWeek.indexOf("Sunday")); // 6



List<String> weekDays = daysOfWeek.subList(0, 5);

System.out.println(weekDays); // [Monday, Tuesday, Wednesday, Thursday, Friday]
```

Since it is **immutable,** only methods that do not change the elements in the list will work. Others will throw an exception.

```
daysOfWeek.set(0, "Funday"); // throws UnsupportedOperationException
daysOfWeek.add("Holiday");   // throws UnsupportedOperationException
```

This situation clearly demonstrates when immutable lists are needed. It's hard to imagine that someone renames a day or adds another one!

> Be careful when working with immutable lists. Sometimes even experienced developers get `UnsupportedOperationException`.

Prior to Java 9, another way to create unmodifiable lists was the following:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

To use it, the package `java.util.Arrays` must be imported.

§3. Mutable lists

When you need to use a **mutable** list, you can take one of two commonly used mutable implementations of the `List` interface.

One of them is familiar to you: the `ArrayList<E>` class. It represents a resizable array. In addition to implementing the `List` interface, it provides methods to manipulate the size of the array that is used internally. These methods are not needed in programs often, so it is better to use an object of this class through the `List` interface.

```
List<Integer> numbers = new ArrayList<>();

numbers.add(15);
numbers.add(10);
numbers.add(20);

System.out.println(numbers); // [15, 10, 20]

numbers.set(0, 30); // no exceptions here



System.out.println(numbers); // [30, 10, 20]
```

If you have an immutable list, you can take the mutable version from it using the following code:

```
List<String> immutableList = Arrays.asList("one", "two", "three");
List<String> mutableList = new ArrayList<>(immutableList);
```

Another mutable implementation of the `List` interface is the `LinkedList` class. It represents a **doubly-linked list** based on connected nodes. All operations that index into the list will traverse the list from the beginning or from the end, whichever is closer to the specified index.

```
List<Integer> numbers = new LinkedList<>();

numbers.add(10);
numbers.add(20);
numbers.add(30);

System.out.println(numbers); // [10, 20, 30]
```

Access to the first and the last element of the list is always carried out in constant time `O(1)` because links are permanently stored in the first and the last element, so adding an item to the end of the list does not mean that you have to iterate the whole list in search of the last element. But accessing/setting an element by its index takes `O(n)` time for a linked list.

> In the general case, `LinkedList` loses to `ArrayList` in memory consumption and speed of operations. But it depends on the problem you are trying to solve.

§4. Iterating over a list

There are no problems to iterate over elements of a list.

```
List<String> names = List.of("Larry", "Kenny", "Sabrina");
```

1) Using the "for-each" loop:

```
// print every name
for (String name : names) {
    System.out.println(name);
}
```

2) Using indexes and the `size()` method:

```
// print every second name
for (int i = 0; i < names.size(); i += 2) {
    System.out.println(names.get(i));
}
```

When you need to go through all elements of a list, we recommend choosing the first way to iterate. The second way is good when you need to skip some elements based on their positions in the list.

§5. List equality

The final question is how lists are compared. Two lists are equal when they contain the same elements in the same order. The equality does not depend on the types of the lists themselves (`ArrayList`, `LinkedList` or something else).

```
Objects.equals(List.of(1, 2, 3), List.of(1, 2, 3));    // true
Objects.equals(List.of(1, 2, 3), List.of(1, 3, 2));    // false
Objects.equals(List.of(1, 2, 3), List.of(1, 2, 3, 1)); // false

List<Integer> numbers = new ArrayList<>();

numbers.add(1);
numbers.add(2);
numbers.add(3);



Objects.equals(numbers, List.of(1, 2, 3)); // true
```

With this, we are finishing our discussion of the `List` interface and common features for all lists. There was a lot of theory. If there's something you do not yet understand, try to practice and go back to the theory when questions arise.