Theory: ArrayList

25 minutes 0 / 5 problems solved

§1. Resizable arrays

One of the most widely used classes of Java Class Library is a class named `ArrayList` that represents a resizable array of objects of a specified type. Unlike the standard array denoted as `[]`, it can dynamically grow after the addition and shrink after the removal of its elements. This behavior is very useful if you do not know the size of the array in advance or you need one that can change sizes over the lifetime of a program.

In fact, this class is built on top of a standard Java array, extending it with a set of convenient operations. Like a standard array, it allows getting the current number of elements (its size) as well as accessing its elements by their indexes.

There is only one restriction: `ArrayList`: being a generic class, it cannot store primitive types. However, it can store any reference types, including `String`'s, wrapper classes (like `Integer`'s), other `ArrayList`'s, and custom classes.

§2. Creating an instance of ArrayList

To start using the class by its short name, make the following import:

```
1 import java.util.ArrayList;
```

Let's consider several ways to create instances of this class.

1) The simplest way is to use a no-argument constructor:

```
1 ArrayList<String> list = new ArrayList<>();
```

The created list is empty, but its initial capacity is 10 (by default).

2) We can also specify the initial capacity of it:

```
1 ArrayList<String> list = new ArrayList<>(50);
```

This list is empty, but its initial capacity is set to 50.

3) Or you can construct an `ArrayList` that consists of elements of another list:

```
1 ArrayList<String> list = new ArrayList<>(anotherList);
```

Regardless of how you create an instance of `ArrayList`, its size will dynamically change. In this lesson, we will create a list with the default capacity like in the first example.

> If you are an advanced user, you know that it is better to create and use an `ArrayList` via its `List` interface. We will do it in the next lessons after learning **inheritance**. We believe that the current approach is enough for now since it requires less knowledge to start using dynamic collections.

§3. Basic methods

The collection has a set of convenient methods that emulate and extend the functionality of standard arrays. Let's discuss what they are. First, let's initialize some collection:

```
1 ArrayList<String> names = new ArrayList<>
(); // empty collection of strings
```

Current topic:

ArrayList

Topic depends on:

- ✗ Dynamic array
- ✓ Iterating over arrays
- ✗ What are collections

Topic is required for:

- List
- Trees in Java
- Rest controller

Table of contents:

First of all, there's a method to determine the size of the collection `size` that return the number of elements of the list. Let's try learning the size of ours:

```
System.out.println(names.size()); // 0
```

As expected, it is empty and the result is zero. We also might want to learn the value of the specified position of the object. For that, collections have `get(int index)` method that returns the object of the list which is present at the specified index.

Next, there are a bunch of methods to add elements and set values of a collection:

- `add(Object o)` adds a passed element to the last position of the collection;
- `add(int index, Object o)` adds a passed element to the specified position of the collection;
- `set(int index, Object o)` replaces the element present at the specified index with the object;

Let's add some names to our collection:

```
names.add("Justin");      // [Justin]
names.add("Helen");       // [Justin, Helen]
names.add(1, "Joshua");   // [Justin, Joshua, Helen]
names.add(0, "Laura");    // [Laura, Justin, Joshua, Helen]
```

And replace one name with another:

```
names.set(3, "Marie"); // now: [Laura, Justin, Joshua, Marie]
```

We can check that everything is as expected:

```
System.out.println(names);        // [Laura, Justin, Joshua, Marie]
System.out.println(names.size()); // 4
System.out.println(names.get(0)); // the first element is "Laura"
System.out.println(names.get(3)); // the last element is "Marie"
```

Finally, there are methods for removing elements from the collection:

- `remove(Object o)` removes the object;
- `remove(int index)` removes **the first occurrence** of the element with a given index;
- `clear()` removes all elements from the collection.

Let's try removing elements by value and index:

```
names.remove("Justin"); // [Laura, Joshua, Marie]
names.remove(1);        // [Laura, Marie]
names.clear();          // []
```

> Important: indexes of elements start with 0 just like for standard arrays

Try to play with this code by yourself and enjoy the power of `ArrayList` .

§4. More ArrayList methods

We've illustrated the possibilities of basic methods for collections in Java applied to an `ArrayList` object. But this class has some more methods of its own. First, let's create another `ArrayList`:

```
/* an ArrayList of Integers, not ints */
ArrayList<Integer> numbers = new ArrayList<>();

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

There's also `addAll(Collection c)` method for adding the whole collection to an `ArrayList`. It appends elements of the provided collection to the end of the list:

```
ArrayList<Integer> numbers2 = new ArrayList<>
();    // creating another list of Integers
numbers2.add(100);
numbers2.addAll(numbers); // [100, 1, 2, 3, 1]
```

The class also has a method called `contains` that checks whether a list contains a value or not, and two methods `indexOf` and `lastIndexOf` that find the index of the first and the last occurrences of an element, respectively. They return `-1` if there is no such index.

Let's see:

```
System.out.println(numbers.contains(2));    // true
System.out.println(numbers.contains(4));    // false
System.out.println(numbers.indexOf(1));     // 0
System.out.println(numbers.lastIndexOf(1)); // 3
System.out.println(numbers.lastIndexOf(4)); // -1
```

As you see, this class provides a rich set of methods to work with elements. You do not have to write them by yourself, as you do for standard arrays.

§5. Iterating over ArrayList

It is possible to iterate over elements of an instance of the class. It is done in the same way as iterating over an array. In the following example, we use *for* and *for-each* loops to add the five first powers of ten in a list and then print the numbers to the standard output.

```
ArrayList<Long> powersOfTen = new ArrayList<>();

int count = 5;
for (int i = 0; i < count; i++) {
    long power = (long) Math.pow(10, i);
    powersOfTen.add(power);
}

for (Long value : powersOfTen) {

    System.out.print(value + " ");

}
```

The code prints the following:

```
1 10 100 1000 10000
```

It is not harder than using a standard array.

§6. Conclusion

We've considered the `ArrayList` class from the `java.util` package. This class is similar to standard Java arrays but has the possibility to dynamically change its size. It has methods to get the size, add, remove and access elements by their indexes. In addition, `ArrayList` provides a set of useful methods that check whether an element is presented in the array and find it. A regular array does not have such methods built-in.

Report a typo

348 users liked this theory. **2** didn't like it. **What about you?**

Start practicing

Comments (15) Hints (0) Useful links (0) Show discussion