Storing Elements in a Collection

Exploring the Collection Interface

The first interface you need to know is the `Collection` interface. It models a plain collection, which can store elements and gives you different ways to retrieve them.

If you want to run the examples in this part, you need to know how to create a collection. We have not covered the `ArrayList` class yet, we will do that later.

Methods That Handle Individual Elements

Let us begin by storing and removing an element from a collection. The two methods involved are `add()` and `remove()`.

- `add(element)`: adds an element in the collection. This method returns a `boolean` in case the operation failed. You saw in the introduction that it should not fail for a `List`, whereas it may fail for a `Set`, because a set does not allow duplicates.
- `remove(element)`: removes the given element from the collection. This method also returns a boolean, because the operation may fail. A remove may fail, for instance, when the item requested for removal is not present in the collection

You can run the following example. Here, you create an instance of the `Collection` interface using the `ArrayList` implementation. The generics used tells the Java compiler that you want to store `String` objects in this collection. `ArrayList` is not the only implementation of `Collection` you may use. More on that later.

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
System.out.println("strings = " + strings);
strings.remove("one");
System.out.println("strings = " + strings);
```

Running the previous code should print the following:

```
strings = [one, two]
strings = [two]
```

You can check for the presence of an element in a collection with the `contains()` method. Note that you can check the presence of any type of element. For instance, it is valid to check for the presence of a `User` object in a collection of `String`. This may seem odd, since there is no chance that this check returns `true`, but it is allowed by the compiler. If you are using an IDE to test this code, your IDE may warn about testing for the presence of a `User` object in a collection of `String` objects.

```java
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
if (strings.contains("one")) {
    System.out.println("one is here");
}
if (!strings.contains("three")) {
    System.out.println("three is not here");
}

User rebecca = new User("Rebecca");
if (!strings.contains(rebecca)) {
    System.out.println("Rebecca is not here");
}
```

Running this code produces the following:

```
one is here
three is not here
Rebecca is not here
```

Methods That Handle Other Collections

This first set of methods you saw allows you to handle individual elements.

There are four such methods: `containsAll()`, `addAll()`, `removeAll()` and `retainAll()`. They define the four fundamental operations on set of objects.

- `containsAll()`: defines the inclusion
- `addAll()`: defines the union
- `removeAll()`: defines the complement
- `retainAll()`: defines the intersection.

The first one is really simple: `containsAll()` takes another collection as an argument and returns `true` if all the elements of the other collections are contained in this collection. The collection passed as an argument does not have to be the same type as this collection: it is legal to ask if a collection of `String`, of type `Collection<String>` is contained in a collection of `User`, of type `Collection<User>`.

Here is an example of the use of this method:

```java
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
strings.add("three");

Collection<String> first = new ArrayList<>();
first.add("one");
first.add("two");

Collection<String> second = new ArrayList<>();
second.add("one");
second.add("four");

System.out.println("Is first contained in strings? " + strings.containsAll(first));
System.out.println("Is second contained in strings? " + strings.containsAll(second));
```

Running this code produces the following:

```
Is first contained in strings? true
Is second contained in strings? false
```

The second one is `addAll()`. It allows you to add all the elements of a given collection to this collection. As with the `add()` method, this may fail for some elements in some cases. This method returns `true` if this collection has been modified by this call. This is an important point to understand: getting a `true` value does not mean that all the elements of the other collection have been added; it means that at least one has been added.

You can see `addAll()` in action on the following example:

```java
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
strings.add("three");

Collection<String> first = new ArrayList<>();
first.add("one");
first.add("four");

boolean hasChanged = strings.addAll(first);

System.out.println("Has strings changed? " + hasChanged);
System.out.println("strings = " + strings);
```

Running this code produces the following result:

```
Has strings changed? true
strings = [one, two, three, one, four]
```

You need to be aware that running this code will produce a different result if you change the implementation of `Collection`. This result stands for `ArrayList`, as you will see in the following, it would not be the same for `HashSet`.

The third one is `removeAll()`. It removes all the elements of this collection that are contained in the other collection. Just as it is the case for `contains()` or `remove()`, the other collection can be defined on any type; it does not have to be compatible with the one of this collection.

You can see `removeAll()` in action on the following example:

```java
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
strings.add("three");

Collection<String> toBeRemoved = new ArrayList<>();
toBeRemoved.add("one");
toBeRemoved.add("four");

boolean hasChanged = strings.removeAll(toBeRemoved);

System.out.println("Has strings changed? " + hasChanged);
System.out.println("strings = " + strings);
```

Running this code produces the following result:

```
Has strings changed? true
strings = [two, three]
```

The last one is `retainAll()`. This operation retains only the elements from this collection that are contained in the other collection; all the others are removed. Once again, as it is the case for `contains()` or `remove()`, the other collection can be defined on any type.

You can see `retainAll()` in action on the following example:

```java
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
strings.add("three");

Collection<String> toBeRetained = new ArrayList<>();
toBeRetained.add("one");
toBeRetained.add("four");

boolean hasChanged = strings.retainAll(toBeRetained);

System.out.println("Has strings changed? " + hasChanged);

```

```
    System.out.println("strings = " + strings);
```

Running this code produces the following result:

```
Has strings changed? true
strings = [one]
```

Methods That Handle The Collection Itself

Then the last batch of methods deal with the collection itself.

You have two methods to check the content of a collection.

- `size()`: Returns the number of elements in a collection, as an `int`.
- `isEmpty()`: Tells you if the given collection is empty or not.

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
if (!strings.isEmpty()) {
    System.out.println("Indeed strings is not empty!");
}
System.out.println("The number of elements in strings is " + strings.size());
```

Running this code produces the following:

```
Indeed strings is not empty!
The number of elements in strings is 2
```

Then you can delete the content of a collection by simply calling `clear()` on it.

```
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
System.out.println("The number of elements in strings is " + strings.size());
strings.clear();
System.out.println("After clearing it, this number is now " + strings.size());
```

Running this code produces the following:

```
The number of elements in strings is 2
After clearing it, this number is now 0
```

Getting an Array of the Elements of a Collection

Even if storing your elements in a collection may make more sense in your application than putting them in an array, there are still cases where getting them in an array is something you will need.

The `Collection` interface gives you three patterns to get the elements of a collection in an array, in the form of three overloads of a `toArray()` method.

The first one is a plain `toArray()` call, with no arguments. This returns your elements in an array of plain objects.

This may not be what you need. If you have a `Collection<String>`, what you could prefer is an array of `String`. You can still cast `Object[]` to `String[]`, but there is no guarantee that this cast will not fail at runtime. If you need type safety, then you can call either of the following methods.

- `toArray(T[] tab)` returns an array or `T`: `T[]`
- `toArray(IntFunction<T[]> generator)`, returns the same type, with a different syntax.

What are the differences between the last two patterns? The first one is readability. Creating an instance of `IntFunction<T[]>` may look weird at first, but writing it with a method reference is really a no brainer.

Here is the first pattern. In this first pattern, you need to pass an array of the corresponding type.

```
Collection<String> strings = ...; // suppose you have 15 elements in that collection

String[] tabString1 = strings.toArray(new String[] {}); // you can pass an empty array
String[] tabString2 = strings.toArray(new String[15]);   // or an array of the right size
```

What is the use of this array passed as an argument? If it is big enough to hold all the elements of the collection, then these elements will be copied in the array, and it will be returned. If there is more room in the array than needed, then first of the unused cell of the array will be set to null. If the array you pass is too small, then a new array of the exact right size is created to hold the elements of the collection.

Here is this pattern in action:

```
Collection<String> strings = List.of("one", "two");

String[] largerTab = {"three", "three", "three", "I", "was", "there"};
System.out.println("largerTab = " + Arrays.toString(largerTab));

String[] result = strings.toArray(largerTab);
System.out.println("result = " + Arrays.toString(result));

System.out.println("Same arrays? " + (result == largerTab));
```

Running the previous code will give you:

```
largerTab = [three, three, three, I, was, there]
result = [one, two, null, I, was, there]

```

```
    Same arrays? true
```

You can see that the array was copied in the first cells of the argument array, and `null` was added right after it, thus leaving the last elements of this array untouched. The returned array is the same array as the one you gave as an argument, with a different content.

Here is a second example, with a zero-length array:

```java
Collection<String> strings = List.of("one", "two");

String[] zeroLengthTab = {};
String[] result = strings.toArray(zeroLengthTab);

System.out.println("zeroLengthTab = " + Arrays.toString(zeroLengthTab));
System.out.println("result = " + Arrays.toString(result));
```

Running this code gives you the following result:

```
zeroLengthTab = []
result = [one, two]
```

Not the same array what we have passed.

A new array has been created in this case.

The second pattern is written using a constructor method reference to implement `IntFunction<T[]>`:

```java
Collection<String> strings = ...;

String[] tabString3 = strings.toArray(String[]::new);
```

In that case, a zero-length array of the right type is created with this function, and this method then calls to `toArray()` with this array passed as an argument.

This pattern of code was added in JDK 8 to improve the readability of the `toArray()` calls.

Filtering out Elements of a Collection with a Predicate

Java SE 8 added a new feature the `Collection` interface: the possibility to filter out elements of a collection with a predicate.

Suppose you have a `List<String>` and you need to remove all the null strings, the empty strings and the strings longer than 5 characters. In Java SE 7 and earlier, you can use the `Iterator.remove()` method to do that, calling it in an `if` statement. You will see this pattern along with the `Iterator` interface. With `removeIf()`, your code becomes much simpler:

```java
Predicate<String> isNull = Objects::isNull;
Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNullOrEmpty = isNull.or(isEmpty);
```

```java
Collection<String> strings = new ArrayList<>();
strings.add(null);
strings.add("");
strings.add("one");
strings.add("two");
strings.add("");
strings.add("three");
strings.add(null);

System.out.println("strings = " + strings);
strings.removeIf(isNullOrEmpty);
System.out.println("filtered strings = " + strings);
```

Running this code produces the following result:

```
strings = [null, , one, two, , three, null]
filtered strings = [one, two, three]
```

Once again, using this method will greatly improve the readability and expressiveness of your application code.

Choosing an Implementation for the Collection Interface

In all these examples, we used `ArrayList` to implement the `Collection` interface.

The fact is: the Collections Framework does not provide a direct implementation of the `Collection` interface. `ArrayList` implements `List`, and because `List` extends `Collection`, it also implements `Collection`.

If you decide to use the `Collection` interface to model the collections in your application, then choosing `ArrayList` as you default implementation is your best choice, most of the time. You will see more discussions on the right implementation to choose in later in this tutorial.

Last update: *September 14, 2021*

Getting to Know the Collection Hierarchy ← **Storing Elements in a Collection** → Iterating over the Elements of a Collection

Home > Tutorials > The Collections Framework > Storing Elements in a Collection