Java Collections - Tutorial

Lars Vogel (c) 2008, 2016 vogella GmbH version 2.8, 29.09.2016
Table of Contents

Introduction to the Java Collections framework. This tutorial explains the usage of the Java collections, e.g. Lists, ArrayLists and Maps with Java 8.

1. Java Collections

1.1. What are collections

The Java language supports arrays to store several objects. An array is initialized with an predefined size during instantiation. To support more flexible data structures the core Java library provides the `collection` framework. A collection is a data structure which contains and processes a set of data. The data stored in the collection is encapsulated and the access to the data is only possible via predefined methods. For example the developer can add elements to an collection via a method. Collections use internally arrays for there storage but hide the complexity of managing the dynamic size from the developer.

For example if your application saves data in an object of type `People`, you can store several `People` objects in a collection.

1.2. Important default implementations

Typical collections are: stacks, queues, deques, lists and trees.

Java typically provides an interface, like `List` and one or several implementations for this interface, e.g., the `ArrayList` class and the `LinkedList` are implementations of the `List` interface.

1.3. Type information with generics

A class or interface whose declaration has one or more type parameters is a generic class or interface. For example `List` defines one type parameter `List<E>`.

Java collections should get parameterized with an type declaration. This enables the Java compiler to check if you try to use your collection with the correct type of objects. *Generics*

allow a type or method to operate on objects of various types while providing compile-time type safety. Before generics you had to cast every object you read from a collection and if you inserted an object of the wrong type into a collection you would create a runtime exception.

1.4. Collections and lambdas

The collection library support lambdas expressions. Operations on collections have been largely simplified with this.

The following code shows an example how to create a Collection of type `List` which is parameterized with `<String>` to indicate to the Java compiler that only Strings are allowed in this list. Is uses a method reference and the `foreach` loop from Java 8.

```
package collections;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class MyArrayList {

    public static void main(String[] args) {

        // create a list using the
        List<String> list = Arrays.asList("Lars", "Simon");

        // alternatively
        List<String> anotherList = new ArrayList<>();
        anotherList.add("Lars");
        anotherList.add("Simon");


        // print each element to the console using method references
        list.forEach(System.out::println);
        anotherList.forEach(System.out::println);

    }
}
```

If you try to put an object into this list which is not an object, you would receive a compiler error.

2. List implementations

2.1. The List interface

The `List` interface is the base interface for collections which allows to store objects in a resizable container.

2.2. ArrayList and LinkedList as implementations

`ArrayList` is an implementation of this interface and allow that elements are dynamically added and removed from the list. If more elements are added to ArrayList than its initial size, its size is increased dynamically. The elements in an `ArrayList` can be accessed directly and

efficiently by using the `get()` and `set()` methods, since ArrayList is implemented based on an array. `ArrayList` is typically used in implementations as implementation class for the `List` interface.

`LinkedList` is implemented as a double linked list. Its performance on `add()` and `remove()` is better than the performance of Arraylist. The `get()` and `set()` methods have worse performance than the ArrayList, as the `LinkedList` does not provide direct access to its members.

The following code demonstrates the usage of `List` and `ArrayList`.

```
package com.vogella.java.collections.list;
import java.util.ArrayList;
import java.util.List;

public class ListExample {
    public static void main(String[] args) {
        // use type inference for ArrayList
        List<Integer> list = Arrays.asList(3,2,1,4,5,6,6);

        // alternative you can declare the list via:
        // List<Integer> list = new ArrayList<>();
        // and use list.add(element); to add elements
        for (Integer integer : list) {
            System.out.println(integer);
        }
    }

}
```

2.3. Sorting lists

You can sort lists using their natural order of via lambdas for defining the `Comparator.compare()`. Typically in Java 8 you use a lambda expression or a method reference for the definion of the `compare` method.

```
package com.vogella.java.collections.list;

import java.util.ArrayList;
import java.util.List;

public class ListSorter {
    public static void main(String[] args) {
        System.out.println("Sorting with natural order");
        List<String> l1 = createList();
        l1.sort(null);
        l1.forEach(System.out::println);

        System.out.println("Sorting with a lamba expression for the
comparison");
        List<String> l2 = createList();
        l2.sort((s1, s2) -> s1.compareToIgnoreCase(s2));  // sort ignoring
case
        l2.forEach(System.out::println);

        System.out.println("Sorting with a method references");
        List<String> l3 = createList();
```

```
        l3.sort(String::compareToIgnoreCase);
        l3.forEach(System.out::println);

    }

    private static List<String>  createList() {
        return Arrays.asList("iPhone", "Ubuntu", "Android", "Mac OS X");
    }

}
```

2.4. Remove list members based on condition

The `removeIf` method allows to remove list items based on a condition.

```
package com.vogella.java.collections.list;

import java.util.ArrayList;
import java.util.List;

public class RemoveIfList {
    public static void main(String[] args) {
        System.out.println("Demonstration of removeIf");
        List<String> l1 = createList();
        // remove all items which contains an "x"
        l1.removeIf(s-> s.toLowerCase().contains("x"));
        l1.forEach(s->System.out.println(s));
    }

    private static List<String>  createList() {
        return Arrays.asList("iPhone", "Ubuntu", "Android", "Mac OS X");
    }
}
```

3. Maps implementations

3.1. Map and HashMap

The `Map` interface defines an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. The `HashMap` class is an efficient implementation of the `Map` interface. The following code demonstrates its usage.

```
package com.vogella.java.collections.map;

import java.util.HashMap;
import java.util.Map;

public class MapTester {
    public static void main(String[] args) {

        // keys are Strings
        // objects are also Strings
        Map<String, String> map = new HashMap<>();
        fillData(map);

        // write to command line
        map.forEach((k, v) -> System.out.printf("%s %s%n", k, v));
```

```
        // add and remove from the map
        map.put("iPhone", "Created by Apple");
        map.remove("Android");

        // write again to command line
        map.forEach((k, v) -> System.out.printf("%s %s%n", k, v));

    }

    private static void fillData(Map<String, String> map) {
        map.put("Android", "Mobile");
        map.put("Eclipse IDE", "Java");
        map.put("Eclipse RCP", "Java");
        map.put("Git", "Version control system");

    }

}
```

3.2. Convert keys in a Map to an array or a list

You can convert your keys or values to an array or list. The following code demonstrates that.

```
package com.vogella.java.collections.map;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ConvertMapTester {
    public static void main(String[] args) {

        // keys are Strings
        // objects are also Strings
        Map<String, String> map = new HashMap<>();
        fillData(map);

        // convert keys to array
        String[] strings = keysAsArray(map);
        for (String string : strings) {
            System.out.println(string);
        }
        // convert keys to list
        List<String> list = keysAsList(map);
        for (String string : list) {
            System.out.println(string);
        }
    }

    private static void fillData(Map<String, String> map) {
        map.put("Android", "Mobile");
        map.put("Eclipse IDE", "Java");
        map.put("Eclipse RCP", "Java");
        map.put("Git", "Version control system");

    }

    private static String[] keysAsArray(Map<String, String> map) {
```

```
        return map.keySet().toArray(new String[map.keySet().size()]);
    }

    // assumes the key is of type String
    private static List<String> keysAsList(Map<String, String> map) {
        List<String> list = new ArrayList<String>(map.keySet());
        return list;
    }

}
```

3.3. Processing every element in the map

To process every element a map you can use the `forEach` method, which take a lambda as parameter.

3.4. Getting the current value or a default for a map

Java 8 introduces the `getOrDefault()` method, which allows to get the current value and if this is not found in the map to return a default value.

```
package com.vogella.java.collections.map;

import java.util.HashMap;
import java.util.Map;

public class MapOrDefault {
    public static void main(String[] args) {

        Map<String,Integer> map = createMap();
        map.put("Android", 1 + map.getOrDefault("Android", 0));

        // write to command line
        map.forEach((k, v) -> System.out.printf("%s %s%n", k, v));
    }

    private static Map<String, Integer> createMap() {
        Map<String, Integer> map = new HashMap<>();
        map.put("Eclipse IDE", 0);
        map.put("Eclipse RCP", 0);
        map.put("Git", 0);
        return map;
    }
}
```

If you want to add a new entry automatically to a map if it is not present you can use the `computeIfAbsent` method to calculate a value and add it to the map.

```
package com.vogella.java.collections.map;

import java.util.HashMap;
import java.util.Map;

public class ComputeIfAbsent {
    public static void main(String[] args) {

        Map<String,Integer> map = createMap();
```

```
        Integer calculatedVaue = map.computeIfAbsent("Java", it -> 0);
        System.out.println(calculatedVaue);

        // write to command line
        map.keySet().forEach(
                key -> System.out.println(key + " " + map.get(key)));
    }

    private static Map<String, Integer> createMap() {
        Map<String, Integer> map = new HashMap<>();
        map.put("Eclipse IDE", 0);
        map.put("Eclipse RCP", 0);
        map.put("Git", 0);
        map.put("Groovy", 0);
        return map;
    }
}
```

4. Useful collection methods

The `java.util.Collections` class provides useful functionalities for working with collections.

Table 1. Collections

Method	Description
Collections.copy(list, list)	Copy a collection to another
Collections.reverse(list)	Reverse the order of the list
Collections.shuffle(list)	Shuffle the list
Collections.sort(list)	Sort the list

5. Using Collections.sort and Comparator in Java

Sorting a collection in Java is easy, just use the `Collections.sort(Collection)` to sort your values. The following code shows an example for this.

```
package collections;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Simple {
    public static void main(String[] args) {
        // create a new ArrayList with the Arrays.asList helper method
        List<Integer> list = Arrays.asList(5,4,3,7,2,1);
        // sort it
        Collections.sort(list);
        // print each element to the console
        list.forEach(System.out::println);
    }
}
```

This is possible because Integer implements the `Comparable` interface. This interface defines the method compare which performs pairwise comparison of the elements and returns -1 if the element is smaller than the compared element, 0 if it is equal and 1 if it is larger.

If what to sort differently you can define your custom implementation based on the `Comparator` interface via a lambda expression.

```
package collections;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class ListCustomSorterExample {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(5,4,3,7,2,1);

        // custom comparator
        Collections.sort(list, (o1, o2) -> (o1>o2 ? -1 : (o1==o2 ? 0 :
1)));
        // alternative can could reuse the integer comparator
        // Collections.sort(list, Integer::);
        list.forEach(System.out::println);
    }

}
```

You can sort by any any attribute or even a combination of attributes. For example if you have objects of type `Person` with the attributes called `income` and `dataOfBirth` you could define different implementations of `Comparator` and sort the objects according to your needs.

6. Exercise: Use Java Collections

Create a new Java project called `com.vogella.java.collections`. Also add a package with the same name.

Create a Java class called *Server* with one String attribute called `url`.

```
package com.vogella.java.collections;

public class Server {
    private String url;
}
```

Create getter and setter methods for this attribute using code generation capabilities of Eclipse. For this select Source ▸ Generate Getters and Setters from the Eclipse menu.

Create via Eclipse a constructor which gets a url as parameter. For this select Source ▸ Generate Constructor using Fields… from the Eclipse menu.

Type `main` in the class body and use code completion (`Ctrl+Space`) to generate a `main` method.

```
*Server.java
1  package com.vogella.java.collections;
2
3  public class Server {
4      private String url;
5
6      public Server(String url) {
7          this.url = url;
8      }
9
10     public String getUrl() {
11         return url;
12     }
13
14     public void setUrl(String url) {
15         this.url = url;
16     }
17
18     main
19 }
20
main - main method
public static void m
}
```

In your `main` method create a `List` of type `ArrayList` and add 3 objects of type `Server` objects to this list.

```
public static void main(String[] args) {
    List<Server> list = new ArrayList<Server>();
    list.add(new Server("http://www.vogella.com"));
    list.add(new Server("http://www.google.com"));
    list.add(new Server("http://www.heise.de"));
}
```

Use code completion to create a *foreach* loop and write the `toString` method to the console. Use code completion based on `syso` for that.

Run your program.

Use Eclipse to create a `toString` method based on the `url` parameter and re-run your program again.