


The Map interface

Theory

Practice

 100% completed, 0 problems solved

▼

Theory

🕒 22 minutes reading

Unskip this topic

Start practicing

In some situations, you need to store pairs of associated objects. For example, when counting the number of words in a text, the first one is a word and the second one is the number of its occurrences in the text. There is a special type of collections called **map** to effectively store such pairs of objects.

A **map** is a collection of key-value pairs. Keys are always unique while values can repeat.

A good example of a map from the real world is a phone book where keys are names of your friends and values are phone numbers associated with them.

```
1  Keys   : Values
2  -----
3  Bob    : +1-202-555-0118
4  James  : +1-202-555-0220
5  Katy   : +1-202-555-0175
```

Maps have some similarities with sets and arrays;

- **keys** of a map form a **set**, but each key has an associated value;
- **keys** of a map are similar to **indexes of an array**, but the keys can have any type including integer numbers, strings and so on.

Due to these reasons, you can encounter some kind of *deja vu* effect when learning maps.

Next, all our examples will use string and numbers as keys since using custom classes as types of keys have some significant points the same as for sets. It will be considered in other topics.

§1. The Map interface

The Collections Framework provides the `Map<K,V>` interface to represent a **map** as an abstract data type. Here, `K` is a type of keys, and `V` is a type of associated values. The `Map` interface is not a subtype of the `Collection` interface, but maps are often considered as collections since they are part of the framework.

The interface declares a lot of methods to work with maps. Some of the methods are similar to methods of `Collection`, while others are unique to maps.

1) Collection-like methods:


- `int size()` returns the number of elements in the map;
- `boolean isEmpty()` returns `true` if the map does not contain elements and `false` otherwise;
- `void clear()` removes all elements from the map.

We hope, these methods do not need any comments.

2) Keys and values processing:


2 required topics

- ✓



[The Set interface](#)

▼
- ✓




[Hash table](#)

▼

5 dependent topics

- [Grouping collectors](#)

▼
- 

[Thread-safe maps](#)

▼
- [HashMap](#)

▼
- [Collections framework](#)

▼
- [Flyweight](#)

▼

Table of contents:

- 1

[The Map interface](#)
- §1.

[The Map interface](#)
- §2.

[Immutable maps](#)
- §3.

[HashMap](#)
- §4.

[LinkedHashMap](#)
- §5.

[TreeMap](#)
- §6.

[Iterating over maps](#)
- §7.

[Other collections as values](#)
- §8.

[Map equality](#)
- [Discussion](#)

- `V put(K key, V value)` associates the specified `value` with the specified `key` and returns the previously associated value with this `key` or `null` ;
- `V get(Object key)` returns the value associated with the key, or `null` otherwise;
- `V remove(Object key)` removes the mapping for a `key` from the map;
- `boolean containsKey(Object key)` returns `true` if the map contains the specified `key` ;
- `boolean containsValue(Object value)` returns `true` if the map contains the specified `value` .

These methods are similar to the methods of collections, except they process key-value pairs.

3) Advanced methods:

- `V putIfAbsent(K key, V value)` puts a pair if the specified key is not already associated with a value (or is mapped to `null`) and return `null` , otherwise, returns the current value;
- `V getOrDefault(Object key, V defaultValue)` returns the value to which the specified key is mapped, or `defaultValue` if this map contains no mapping for the key.

These methods together with some others are often used in real projects.

4) Methods which return other collections:

- `Set<K> keySet()` Returns a `Set` view of the keys contained in this map;
- `Collection<V> values()` returns a `Collection` view of the values contained in this map;
- `Set<Map.Entry<K, V>> entrySet()` returns a `Set` view of the entries (associations) contained in this map.

This is not even a complete list of methods since `Map` is a really huge interface. The documentation really helps when using maps.

To start using a map, you need to instantiate one of its implementations: `HashMap` , `TreeMap` , and `LinkedHashMap` . They use different rules for ordering elements and have some additional methods. There are also **immutable** maps whose names are not important for programmers.

§2. Immutable maps

The simplest way to create a **map** is to invoke the `of` method of the `Map` interface. The method takes zero or any even number of arguments in the format `key1, value1, key2, value2, ...` and returns an **immutable** map.

```
1 Map<String, String> emptyMap = Map.of();
2
3 Map<String, String> friendPhones = Map.of(
4     "Bob", "+1-202-555-0118",
5     "James", "+1-202-555-0220",
6     "Katy", "+1-202-555-0175"
7 );
```

Now let's consider some operations that can be applied to **immutable** maps using our example with `friendPhones` .

The size of a map equals to the number of pairs contained in it.

```
1 System.out.println(emptyMap.size());    // 0
2 System.out.println(friendPhones.size()); // 3
```

It is possible to get a value from a map by its key:

```

1  String bobPhone = friendPhones.get("Bob"); // +1-202-555-0118
2  String alicePhone = friendPhones.get("Alice"); // null
3  String phone = friendPhones.getOrDefault("Alex", "Unknown phone");
// Unknown phone

```

Note that the `getOrDefault` method provides a simple way to prevent **NPE** since it avoids `null` 's.

It is also possible to check whether a map contains a particular key or value by using `containsKey` and `containsValue` methods.

We can directly access the set of keys and collection of values from a map:

```

1  System.out.println(friendPhones.keySet()); // [James, Bob, Katy]
2  System.out.println(friendPhones.values()); // [+1-202-555-0220, +1-202-555-0118, +1-202-555-0175]

```

Since it is **immutable**, only methods that do not change the elements of this map will work. Others will throw an exception `UnsupportedOperationException`. If you'd like to put or to remove elements, use one of `HashMap`, `TreeMap` or `LinkedHashMap`.

§3. HashMap

The `HashMap` class represents a map backed by a **hash table**. This implementation provides constant-time performance for `get` and `put` methods assuming the hash function disperses the elements properly among the buckets.

The following example demonstrates a map of products where key is the product code and value is the name.

```

1  Map<Integer, String> products = new HashMap<>();
2
3  products.put(1000, "Notebook");
4  products.put(2000, "Phone");
5  products.put(3000, "Keyboard");
6
7  System.out.println(products); // {2000=Phone, 1000=Notebook, 3000=Keyboard}
8
9  System.out.println(products.get(1000)); // Notebook
10
11
12
13  products.remove(1000);
14
15
16
17  System.out.println(products.get(1000)); // null
18
19
20
21  products.putIfAbsent(3000, "Mouse"); // it does not change the
current element
22
23
24
25  System.out.println(products.get(3000)); // Keyboard

```

This implementation is often used in practice since it is highly-optimized for putting and getting pairs.

§4. LinkedHashMap

The `LinkedHashMap` stores the order in which elements were inserted.

Let's see a part of the previous example again:

```
1  Map<Integer, String> products = new LinkedHashMap<>(); // ordered
map of products
2
3  products.put(1000, "Notebook");
4  products.put(2000, "Phone");
5  products.put(3000, "Keyboard");
6
7  System.out.println(products); // it's always ordered {1000=Notebook,
2000=Phone, 3000=Keyboard}
```

In this code, the order of pairs is always the same and matches the order in which they are inserted into the map.

§5. TreeMap

The `TreeMap` class represents a map that gives us guarantees on the order of the elements. It is corresponding to the sorting order of the keys determined either by their natural order (if they implement the `Comparable` interface) or by specific `Comparator` implementation.

This class implements the `SortedMap` interface which extends the base `Map` interface. It provides some new methods, related to comparisons of keys:

- `Comparator<? super K> comparator()` returns the comparator used to order elements in the map or `null` if the map uses the natural ordering of its keys;
- `E firstKey()` returns the first (lowest) key in the map;
- `E lastKey()` returns the last (highest) key in the map;
- `SortedMap<K, V> headMap(K toKey)` returns a submap containing elements whose keys are strictly less than `toKey` ;
- `SortedMap<K, V> tailMap(K fromKey)` returns a submap containing elements whose keys are greater than or equal to `fromKey` ;
- `SortedMap<K, V> subMap(K fromKey, K toKey)` returns a submap containing elements whose keys are in range `fromKey` (inclusive) `toKey` (exclusive);

The example below demonstrates how to create and use an object of `TreeMap` . This map is filled with events, each of them has a date (key) and title (value).

`LocalDate` is a class that represents a date. The invocation of `LocalDate.of(year, month, day)` method creates the specified date object with the given year, month and day.

```
1  SortedMap<LocalDate, String> events = new TreeMap<>();
2
3  events.put(LocalDate.of(2017, 6, 6), "The Java Conference");
4  events.put(LocalDate.of(2017, 6, 7), "Another Java Conference");
5  events.put(LocalDate.of(2017, 6, 8), "Discussion: career or
education?");
6  events.put(LocalDate.of(2017, 6, 9), "The modern art");
7  events.put(LocalDate.of(2017, 6, 10), "Coffee master class");
8
9  LocalDate fromInclusive = LocalDate.of(2017, 6, 8);
1
0  LocalDate toExclusive = LocalDate.of(2017, 6, 10);
```

```

1
1
1
2 System.out.println(events.subMap(fromInclusive, toExclusive));

```

The code outputs the resulting submap:

```

1 {2017-06-08=Discussion: career or education?, 2017-06-09=The modern art}

```

Use `TreeMap` only when you really need the sorting order of elements, since this implementation is less efficient than `HashMap`.

§6. Iterating over maps

It is impossible to directly iterate over a map since it does not implement the `Iterable` interface. Fortunately, some methods of maps return other collections which are iterable. The order of elements when iterating depends on the concrete implementation of the `Map` interface.

The following code shows how to get keys and values in a for-each loop:

```

1 Map<String, String> friendPhones = Map.of(
2     "Bob", "+1-202-555-0118",
3     "James", "+1-202-555-0220",
4     "Katy", "+1-202-555-0175"
5 );
6
7 // printing names
8 for (String name : friendPhones.keySet()) {
9     System.out.println(name);
10 }
11
12
13 // printing phones
14 for (String phone : friendPhones.values()) {
15     System.out.println(phone);
16 }

```

If you want to print a key and its associated value at the same iteration, you can get `entrySet()` and iterate over it.

```

1 for (var entry : friendPhones.entrySet()) {
2     System.out.println(entry.getKey() + ": " + entry.getValue());
3 }

```

This code prints all pairs as we expect:

```

1 Bob: +1-202-555-0118
2 James: +1-202-555-0220
3 Katy: +1-202-555-0175

```

We use `var` released in Java 10 to declare the loop's variable `entry`, but it is not necessary. If you have an older version of Java or just don't want to use `var`, you can write the data type explicitly like `Map.Entry<String, String>`.

The same behavior can be achieved by using a lambda expression with two arguments if you prefer this way:

```
1 friendPhones.forEach((name, phone) -> System.out.println(name + ": " + phone));
```

§7. Other collections as values

It is possible to store other collections as values in maps since collections are objects as well.

Here is an example with a map of synonyms:

```
1 Map<String, Set<String>> synonyms = new HashMap<>();
2
3 synonyms.put("Do", Set.of("Execute"));
4 synonyms.put("Make", Set.of("Set", "Attach", "Assign"));
5 synonyms.put("Keep", Set.of("Hold", "Retain"));
6
7 // {Keep=[Hold, Retain], Make=[Attach, Assign, Set], Do=[Execute]}
8 System.out.println(synonyms);
```

Storing collections as keys of a map, on the other hand, is not a common case and it has some restrictions. Such keys should be represented by **immutable** collections. We will not consider this case here.

§8. Map equality

Two maps are considered equal if they contain the same keys and values. Types of maps are not important.

So, the following maps are fully equal:

```
1 Map<String, Integer> namesToAges1 = Map.of("John", 30, "Alice", 28);
2 Map<String, Integer> namesToAges2 = new HashMap<>();
3
4 namesToAges2.put("Alice", 28);
5 namesToAges2.put("John", 30);
6
7 System.out.println(Objects.equals(namesToAges1, namesToAges2)); //
true
```

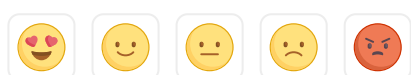
But the following two maps are different since the second map does not include "Alice":

```
1 Map<String, Integer> namesToAges1 = Map.of("John", 30, "Alice", 28);
2 Map<String, Integer> namesToAges2 = Map.of("John", 30);
3
4 System.out.println(Objects.equals(namesToAges1, namesToAges2)); //
false
```

By this, we are finishing our consideration of maps. There was a lot of theory. If there's something you don't understand, try to practice anyway and return to the theory when questions arise.

 Report a typo

553 users liked this piece of theory. **5** didn't like it. **What about you?**



Start practicing


Unskip this topic

This content was created over 5 years ago and updated about 24 hours ago. [Share your feedback below in comments to help us improve it!](#)

Comments (23)

Useful links (1)

Show discussion

 **JetBrains Academy**

Tracks


Pricing

For organizations

About

Contribute

Careers

 Become beta tester

Be the first to see what's new