Theory: Set

39 minutes 0 / 5 problems solved

Skip this topic Start practicing

2887 users solved this topic. Latest completion was about 18 hours ago.

When you need only unique elements within a collection, get rid of duplicates in a sequence, or intend to perform some mathematical operations, you may use a **set**.

A **set** is a collection of unique elements like a mathematical set. A set is significantly different from an array or a list since it's impossible to get an element by its index.

In this topic, we will consider **mutable** and **immutable** sets and their differences. All our examples will use string and numbers since storing objects of custom classes as elements have some significant points. It will be considered in other topics.

§1. The Set interface

The Collections Framework provides the `Set<E>` interface to represent a **set** as an abstract data type. It inherits all the methods from the `Collection<E>` interface, but doesn't add any new ones. The most used methods are the following:

- `contains(Object o)` returns `true` only if this collection contains the specified element;
- `add(E element)` adds a new element to the set;
- `addAll(Collection<E> coll)` adds all of the elements from other collection to this set;
- `retainAll(Collection<E> coll)` retains only the elements in this set that are contained in the specified collection;
- `remove(Object obj)` removes the specified element from this set;
- `removeAll(Collection<E> coll)` removes from this set all elements that are contained in the specified collection;
- `size()` returns the number of elements in this set;
- `isEmpty()` returns `true` only if this collection contains no elements;
- `clear()` removes all elements from this collection

The `add` and `addAll` methods add elements to the set only if those elements are not already in the set. A set always contain only unique elements.

To start using a set, you need to instantiate one of its implementations: `HashSet`, `TreeSet`, and `LinkedHashSet`. These are mutable sets and use different rules for ordering elements and have some additional methods. They are also optimized for different types of operations. There are also **immutable** sets whose names are not important for programmers. They also implement the `Set<E>` interface.

As an addition, there is a high-performance implementation `EnumSet` for `enum` types. We will not consider it in this topic.

§2. Immutable sets

The simplest way to create a **set** is to invoke the `of` method of the `Set` interface.

```
Set<String> emptySet = Set.of();
Set<String> persons = Set.of("Larry", "Kenny", "Sabrina");
Set<Integer> numbers = Set.of(100, 200, 300, 400);
```

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

Current topic:

Set

Topic depends on:

× The Collections Framework overview

Topic is required for:

Iterator and Iterable

Functional data processing with streams

Map

Table of contents:

The order of elements of **immutable** sets is not fixed:

```
System.out.println(emptySet); // []
System.out.println(persons);  // [Kenny, Larry, Sabrina] or another order
System.out.println(numbers);  // [400, 200, 300, 100] or another order
```

One of the most used set's operations is checking whether a set contains an element. Here is an example:

```
System.out.println(emptySet.contains("hello")); // false
System.out.println(persons.contains("Sabrina")); // true
System.out.println(persons.contains("John")); // false
System.out.println(numbers.contains(300)); // true
```

For **immutable** sets, it's only possible to invoke `contains`, `size`, and `isEmpty` methods. All others will throw `UnsupportedOperationException` since they try to change the set. If you'd like to add / remove elements, use one of `HashSet`, `TreeSet` or `LinkedHashSet`.

Next, we will consider three primary **mutable** implementations of the `Set` interface.

§3. HashSet

The `HashSet` class represents a set backed by a **hash table**. It uses hash codes of elements to effectively store them. It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.

The following example demonstrates creating a `HashSet` and adding countries to it (with a duplicate). The output result does not contain duplicates.

```
Set<String> countries = new HashSet<>();

countries.add("India");
countries.add("Japan");
countries.add("Switzerland");
countries.add("Japan");
countries.add("Brazil");

System.out.println(countries); // [Japan, Brazil, Switzerland, India]

System.out.println(countries.contains("Switzerland")); // true
```

You must not rely on the order of elements in this set, even with the **for-each** loop.

The `HashSet` class offers constant time `O(1)` performance for the basic operations (`add`, `remove`, and `contains`), assuming the hash function disperses the elements properly among the buckets.

In practice, sets are often used to check whether some elements belong to them. The `HashSet` class is especially recommended for such cases since its `contains` operation is highly optimized.

§4. TreeSet

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

The order in which the elements would be sorted is the same as if you used a sort algorithm on an array or list containing these elements.

The `TreeSet` class implements the `SortedSet` interface which extends the base `Set` interface. The `SortedSet` interface provides some new methods related to comparisons of elements:

- `Comparator<? super E> comparator()` returns the **comparator** used to order elements in the set or `null` if the set uses the natural ordering of its elements;
- `E first()` returns the first (lowest) element in the set;
- `E last()` returns the last (highest) element in the set;
- `SortedSet<E> headSet(E toElement)` returns a subset containing elements that are strictly less than `toElement`;
- `SortedSet<E> tailSet(E fromElement)` returns a subset containing elements that are greater than or equal to `fromElement`;
- `SortedSet<E> subSet(E fromElement, E toElement)` returns a subset containing elements in the range `fromElement` (inclusive) `toElement` (exclusive).

The following example demonstrates some of the listed methods.

```java
SortedSet<Integer> sortedSet = new TreeSet<>();

sortedSet.add(10);
sortedSet.add(15);
sortedSet.add(13);
sortedSet.add(21);
sortedSet.add(17);

System.out.println(sortedSet); // [10, 13, 15, 17, 21]

System.out.println(sortedSet.headSet(15)); // [10, 13]



System.out.println(sortedSet.first()); // minimum is 10

System.out.println(sortedSet.last());  // maximum is 21
```

Note, `HashSet` is much faster than `TreeSet`: constant-time versus log-time for most operations, it offers no ordering guarantees. If you need to use the operations from the `SortedSet` interface, or if the value-ordered iteration is required, use `TreeSet`, otherwise, `HashSet` would be a better choice.

§5. LinkedHashSet

The `LinkedHashSet` class represents a set with linked elements. It differs from `HashSet` by guaranteeing that the order of the elements is the same as the order they were inserted to the set. Reinserting an element that is already in the `LinkedHashSet` does not change this order.

In some sense, `LinkedHashSet` is something intermediate between `HashSet` and `TreeSet`. Implemented as a hash table with a linked list running through it, this set provides *insertion-ordered* iteration and runs nearly as fast as `HashSet`.

The following example demonstrates this.

```java
Set<Character> characters = new LinkedHashSet<>();

for (char c = 'a'; c <= 'k'; c++) {
    characters.add(c);
}

System.out.println(characters); // [a, b, c, d, e, f, g, h, i, j, k]
```

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

The `LinkedHashSet` implementation spares its clients from the chaotic ordering provided by `HashSet` without incurring the increased time cost of operations associated with `TreeSet`. But `LinkedHashSet` requires more memory for storing elements.

§6. Set operations

You have already seen some operations on sets. Now let's look at operations that are usually called as **set theoretic operations** that come from math. It's funny that in Java they are common for all collections, not only for sets.

Here is an example of such operations. First of all, we create a mutable set. Then, we apply operations to it, changing the elements.

```java
// getting a mutable set from an immutable one
Set<String> countries = new HashSet<>(List.of("India", "Japan", "Switzerland"));

countries.addAll(List.of("India", "Germany", "Algeria"));
System.out.println(countries ); // [Japan, Algeria, Switzerland, Germany, India]

countries.retainAll(List.of("Italy", "Japan", "India", "Germany"));
System.out.println(countries ); // [Japan, Germany, India]

countries.removeAll(List.of("Japan", "Germany", "USA"));

System.out.println(countries ); // [India]
```

After performing `addAll`, the set `countries` does not contain duplicate countries. The `retainAll` and `removeAll` operations affect only those elements which are specified in the passed sets. It is also possible to use any class that implements the `Collection` interface for these methods (e.g. `ArrayList`).

In math and other programming languages, the demonstrated set operations are known as **union** (`addAll`), **intersection** (`retainAll`) and **difference** (`removeAll`).

There is also a method that allows us to check whether a set is a subset of (i.e. contained in) another set.

```java
Set<String> countries = new HashSet<>(List.of("India", "Japan", "Algeria"));

System.out.println(countries.containsAll(Set.of())); // true
System.out.println(countries.containsAll(Set.of("India", "Japan")));   // true
System.out.println(countries.containsAll(Set.of("India", "Germany"))); // false
System.out.println(countries.containsAll(Set.of("Algeria", "India", "Japan"))); // true
```

As you can see, this method returns `true` even for an empty set and a set that is fully equal to the initial set.

§7. Set equality

Last but not least is how sets are compared. Two sets are equal when they contain the same elements. Equality does not depend on the types of sets themselves.

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

Set<Integer> numbers = new HashSet<>();

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



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

We assume that the given examples do not need any comments.

§8. Summary

We finished the consideration of the `Set` interface and common features for all sets. Remember, there are immutable and mutable sets. They can be both unordered and ordered, but always do not contain duplicates.

Report a typo

339 users liked this theory. 6 didn't like it. **What about you?**

Start practicing

Comments (14) Hints (0) Useful links (3) Show discussion