

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.

```
1 Set<String> emptySet = Set.of();
2 Set<String> persons = Set.of("Larry", "Kenny", "Sabrina");
3 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:

[1 Set](#)

[§1. The Set interface](#)

[§2. Immutable sets](#)

[§3. HashSet](#)

[§4. TreeSet](#)

[§5. LinkedHashSet](#)

[§6. Set operations](#)

[§7. Set equality](#)

[§8. Summary](#)

[Feedback & Comments](#)

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

```
1 System.out.println(emptySet); // []
2 System.out.println(persons); // [Kenny, Larry, Sabrina] or another order
3 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:

```
1 System.out.println(emptySet.contains("hello")); // false
2 System.out.println(persons.contains("Sabrina")); // true
3 System.out.println(persons.contains("John")); // false
4 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.

```
1 Set<String> countries = new HashSet<>();
2
3 countries.add("India");
4 countries.add("Japan");
5 countries.add("Switzerland");
6 countries.add("Japan");
7 countries.add("Brazil");
8
9 System.out.println(countries); // [Japan, Brazil, Switzerland, India]
1
0 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.

```
1 SortedSet<Integer> sortedSet = new TreeSet<>();
2
3 sortedSet.add(10);
4 sortedSet.add(15);
5 sortedSet.add(13);
6 sortedSet.add(21);
7 sortedSet.add(17);
8
9 System.out.println(sortedSet); // [10, 13, 15, 17, 21]
10
11 System.out.println(sortedSet.headSet(15)); // [10, 13]
12
13
14
15
16 System.out.println(sortedSet.first()); // minimum is 10
17
18
19 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.

```
1 Set<Character> characters = new LinkedHashSet<>();
2
3 for (char c = 'a'; c <= 'k'; c++) {
4     characters.add(c);
5 }
6
7 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.

```
1 // getting a mutable set from an immutable one
2 Set<String> countries = new HashSet<>
(List.of("India", "Japan", "Switzerland"));
3
4 countries.addAll(List.of("India", "Germany", "Algeria"));
5
System.out.println(countries ); // [Japan, Algeria, Switzerland, Germany, India]
6
7 countries.retainAll(List.of("Italy", "Japan", "India", "Germany"));
8 System.out.println(countries ); // [Japan, Germany, India]
9
10 countries.removeAll(List.of("Japan", "Germany", "USA"));
11
12 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.

```
1 Set<String> countries = new HashSet<>
(List.of("India", "Japan", "Algeria"));
2
3 System.out.println(countries.containsAll(Set.of())); // true
4
System.out.println(countries.containsAll(Set.of("India", "Japan"))); // true
5
System.out.println(countries.containsAll(Set.of("India", "Germany"))); // false
6
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.

```
1 Objects.equals(Set.of(1, 2, 3), Set.of(1, 3)); // false
2 Objects.equals(Set.of(1, 2, 3), Set.of(1, 2, 3)); // true
3 Objects.equals(Set.of(1, 2, 3), Set.of(1, 3, 2)); // true
4
5 Set<Integer> numbers = new HashSet<>();
6
7 numbers.add(1);
8 numbers.add(2);
9 numbers.add(3);
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

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](#)