

16.9 Sets

A `Set` is a collection of unique elements (i.e., no duplicates). The collections framework contains several `Set` implementations, including `HashSet` and `TreeSet`. `HashSet` stores its elements (unordered) in a *hash table*, and `TreeSet` stores its elements (ordered) in a *tree*. Hash tables are presented in [Section 16.10](#). Trees are discussed in [Section 21.7](#). [Figure 16.15](#) uses a `HashSet` to *remove duplicate strings* from a `List`. Recall that both `List` and `Collection` are generic types, so line 14 creates a `List` that contains `String` objects, and line 18 passes the collection to method `printNonDuplicates` (lines 22–33), which takes a `Collection` argument. Line 24 constructs a `HashSet<String>` from the `Collection<String>` argument. By definition, `Sets` do *not* contain duplicates, so when the `HashSet` is constructed, it *removes any duplicates* in the `Collection`. Lines 28–30 output elements in the `Set`.

```
1  // Fig. 16.15: SetTest.java
2  // HashSet used to remove duplicate values from
3      import java.util.List;
4      import java.util.Arrays;
5      import java.util.HashSet;
6      import java.util.Set;
7      import java.util.Collection;
```

```

9    public class SetTest {
10   public static void main(String[] args) {
11       // create and display a List<String>
12       String[] colors = {"red", "white", "blue",
13                           "orange", "tan", "white", "cyan", "peac
14       List<String> list = Arrays.asList(colors);
15       System.out.printf("List: %s\n", list);
16
17       // eliminate duplicates then print the uni
18       printNonDuplicates(list);
19   }
20
21   // create a Set from a Collection to eliminat
22   private static void printNonDuplicates(Collec
23       // create a HashSet
24       Set<String> set = new HashSet<>(values);
25
26       System.out.printf("%nNonduplicates are: ")
27
28       for (String value : set) {
29           System.out.printf("%s ", value);
30       }
31
32       System.out.println();
33   }
34 }

```

List: [red, white, blue, green, gray, orange, tan, wh
orange]

Nonduplicates are: tan green peach cyan red orange gr

Fig. 16.15

HashSet used to remove duplicate values from an array of strings.

Sorted Sets

The collections framework also includes the `SortedSet` **interface** (which extends `Set`) for sets that maintain their elements in *sorted* order—either the *elements' natural order* (e.g., numbers are in *ascending* order) or an order specified by a `Comparator`. Class `TreeSet` implements `SortedSet`. The program in [Fig. 16.16](#) places `Strings` into a `TreeSet`. The `Strings` are sorted as they're added to the `TreeSet`. This example also demonstrates *rangeview* methods, which enable a program to view a portion of a collection.

```
1 // Fig. 16.16: SortedSetTest.java
2 // Using SortedSets and TreeSets.
3 import java.util.Arrays;
4 import java.util.SortedSet;
5 import java.util.TreeSet;
6
7 public class SortedSetTest {
8     public static void main(String[] args) {
9         // create TreeSet from array colors
10        String[] colors = {"yellow", "green", "blue",
11                           "white", "orange", "red", "green"};
12        SortedSet<String> tree = new TreeSet<>(Arrays.asList(colors));
13
14        System.out.print("sorted set: ");
15        printSet(tree);
16
17        // get headSet based on "orange"
18        System.out.print("headSet (<\"orange\")<: ")
```

```
19      printSet(tree.headSet("orange"));
20
21      // get tailSet based upon "orange"
22      System.out.print("tailSet (\"orange\"): ")
23      printSet(tree.tailSet("orange"));
24
25      // get first and last elements
26      System.out.printf("first: %s\n", tree.first());
27      System.out.printf("last : %s\n", tree.last());
28  }
29
30  // output SortedSet using enhanced for statement
31  private static void printSet(SortedSet<String> set) {
32      for (String s : set) {
33          System.out.printf("%s ", s);
34      }
35
36      System.out.println();
37  }
38  }
```

```
sorted set: black green grey orange red tan white yellow
headSet ("orange"): black green grey
tailSet ("orange"): orange red tan white yellow
first: black
last : yellow
```

Fig. 16.16

Using SortedSets and TreeSets.

Line 12 creates a `TreeSet<String>` that contains the

elements of array `colors`, then assigns the new `TreeSet<String>` to `SortedSet<String>` variable `tree`. Line 15 outputs the initial set of strings using method `printSet` (lines 31–37), which we discuss momentarily. Line 19 calls `TreeSet` **method** `headSet` to get a subset of the `TreeSet` in which every element is less than `"orange"`. The view returned from `headSet` is then output with `printSet`. If any changes are made to the subset, they'll *also* be made to the original `TreeSet`, because the subset returned by `headSet` is a view of the `TreeSet`.

Line 23 calls `TreeSet` **method** `tailSet` to get a subset in which each element is greater than or equal to `"orange"`, then outputs the result. Any changes made through the `tailSet` view are made to the original `TreeSet`. Lines 26–27 call `SortedSet` methods `first` and `last` to get the smallest and largest elements of the set, respectively.

Method `printSet` (lines 31–37) accepts a `SortedSet` as an argument and prints it. Lines 32–34 print each element of the `SortedSet` using the enhanced `for` statement.