

Chapter 16

Generic Collections

Java How to Program, 11/e, Global Edition
Questions? E-mail paul.deitel@deitel.com

OBJECTIVES

In this chapter you'll:

- Learn what collections are.
- Use class **Arrays** for array manipulations.
- Learn the type-wrapper classes that enable programs to process primitive data values as objects.
- Understand the boxing and unboxing that occurs automatically between objects of the type-wrapper classes and their corresponding primitive types.

OBJECTIVES (cont.)

- Use prebuilt generic data structures from the collections framework.
- Use various algorithms of the `Collections` class to process collections.
- Use iterators to “walk through” a collection.
- Learn about synchronization and modifiability wrappers.
- Learn about Java SE 9’s new factory methods for creating small immutable `Lists`, `Sets` and `Maps`.

16.1 Introduction

16.2 Collections Overview

16.3 Type-Wrapper Classes

16.4 Autoboxing and Auto-Unboxing

16.5 Interface Collection and Class
Collections

16.6 Lists

16.6.1 `ArrayList` and `Iterator`

16.6.2 `LinkedList`

16.7 Collections Methods

16.7.1 Method `sort`

16.7.2 Method `shuffle`

16.7.3 Methods `reverse`, `fill`, `copy`, `max` and `min`

16.7.4 Method `binarySearch`

16.7.5 Methods `addAll`, `frequency` and `disjoint`

16.8 Class PriorityQueue and Interface Queue

16.9 Sets

16.10 Maps

16.11 Synchronized Collections

16.12 Unmodifiable Collections

16.13 Abstract Implementations

16.14 Java SE 9: Convenience Factory Methods for Immutable Collections

16.15 Wrap-Up

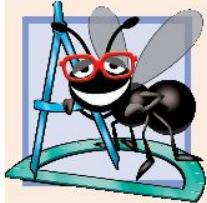
Interface	Description
Collection	The root interface in the collections hierarchy from which interfaces Set , Queue and List are derived.
Set	A collection that does <i>not</i> contain duplicates.
List	An ordered collection that <i>can</i> contain duplicate elements.
Map	A collection that associates keys to values and <i>cannot</i> contain duplicate keys. Map does not derive from Collection .
Queue	Typically a <i>first-in, first-out</i> collection that models a <i>waiting line</i> ; other orders can be specified.

Fig. 16.1 | Some collections-framework interfaces.



Good Programming Practice 16.1

Avoid reinventing the wheel—rather than building your own data structures, use the interfaces and collections from the Java collections framework, which have been carefully tested and tuned to meet most application requirements.



Software Engineering Observation 16.1

Collection is used commonly as a parameter type in methods to allow polymorphic processing of all objects that implement interface Collection.



Software Engineering Observation 16.2

Most collection implementations provide a constructor that takes a `Collection` argument, thereby allowing a new collection to be constructed containing the elements of the specified collection.

```
1 // Fig. 16.2: CollectionTest.java
2 // Collection interface demonstrated via an ArrayList object.
3 import java.util.List;
4 import java.util.ArrayList;
5 import java.util.Collection;
6 import java.util.Iterator;
7
8 public class CollectionTest {
9     public static void main(String[] args) {
10         // add elements in colors array to list
11         String[] colors = {"MAGENTA", "RED", "WHITE", "BLUE", "CYAN"};
12         List<String> list = new ArrayList<String>();
13
14         for (String color : colors) {
15             list.add(color); // adds color to end of list
16         }
17     }
}
```

Fig. 16.2 | Collection interface demonstrated via an ArrayList object. (Part I of 4.)

```
18 // add elements in removeColors array to removeList
19 String[] removeColors = {"RED", "WHITE", "BLUE"};
20 List<String> removeList = new ArrayList<String>();
21
22 for (String color : removeColors) {
23     removeList.add(color);
24 }
25
26 // output list contents
27 System.out.println("ArrayList: ");
28
29 for (int count = 0; count < list.size(); count++) {
30     System.out.printf("%s ", list.get(count));
31 }
```

Fig. 16.2 | Collection interface demonstrated via an ArrayList object. (Part 2 of 4.)

```
32
33     // remove from list the colors contained in removeList
34     removeColors(list, removeList);
35
36     // output list contents
37     System.out.printf("%n%nArrayList after calling removeColors:%n");
38
39     for (String color : list) {
40         System.out.printf("%s ", color);
41     }
42 }
43
```

Fig. 16.2 | Collection interface demonstrated via an `ArrayList` object. (Part 3 of 4.)

```
44     // remove colors specified in collection2 from collection1
45     private static void removeColors(Collection<String> collection1,
46             Collection<String> collection2) {
47         // get iterator
48         Iterator<String> iterator = collection1.iterator();
49
50         // loop while collection has items
51         while (iterator.hasNext()) {
52             if (collection2.contains(iterator.next())) {
53                 iterator.remove(); // remove current element
54             }
55         }
56     }
57 }
```

ArrayList:

MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:

MAGENTA CYAN

Fig. 16.2 | Collection interface demonstrated via an ArrayList object. (Part 4 of 4.)



Common Programming Error 16.1

If a collection is modified by one of its methods after an iterator is created for that collection, the iterator immediately becomes invalid—any operation performed with the iterator fails immediately and throws a `ConcurrentModificationException`. For this reason, iterators are said to be “fail fast.” Fail-fast iterators help ensure that a modifiable collection is not manipulated by two or more threads at the same time, which could corrupt the collection. In Chapter 23, Concurrency, you’ll learn about concurrent collections (package `java.util.concurrent`) that can be safely manipulated by multiple concurrent threads.



Software Engineering Observation 16.3

We refer to the `ArrayLists` in this example via `List` variables. This makes our code more flexible and easier to modify—if we later determine that `LinkedLists` would be more appropriate, only the lines where we created the `ArrayList` objects (lines 12 and 20) need to be modified. In general, when you create a collection object, refer to that object with a variable of the corresponding collection interface type. Similarly, implementing method `removeColors` to receive `Collection` references enables the method to be used with any collection that implements the interface `Collection`.

```
1 // Fig. 16.3: ListTest.java
2 // Lists, LinkedLists and ListIterators.
3 import java.util.List;
4 import java.util.LinkedList;
5 import java.util.ListIterator;
6
7 public class ListTest {
8     public static void main(String[] args) {
9         // add colors elements to list1
10        String[] colors =
11            {"black", "yellow", "green", "blue", "violet", "silver"};
12        List<String> list1 = new LinkedList<>();
13
14        for (String color : colors) {
15            list1.add(color);
16        }
17    }
```

Fig. 16.3 | Lists, LinkedLists and ListIterators. (Part I of 5.)

```
18     // add colors2 elements to list2
19     String[] colors2 =
20         {"gold", "white", "brown", "blue", "gray", "silver"};
21     List<String> list2 = new LinkedList<>();
22
23     for (String color : colors2) {
24         list2.add(color);
25     }
26
27     list1.addAll(list2); // concatenate lists
28     list2 = null; // release resources
29     printList(list1); // print list1 elements
30
31     convertToUppercaseStrings(list1); // convert to uppercase string
32     printList(list1); // print list1 elements
33
34     System.out.printf("%nDeleting elements 4 to 6..."); 
35     removeItems(list1, 4, 7); // remove items 4-6 from list
36     printList(list1); // print list1 elements
37     printReversedList(list1); // print list in reverse order
38 }
```

Fig. 16.3 | Lists, LinkedLists and ListIterators. (Part 2 of 5.)

```
39
40     // output List contents
41     private static void printList(List<String> list) {
42         System.out.printf("%nlist:%n");
43
44         for (String color : list) {
45             System.out.printf("%s ", color);
46         }
47
48         System.out.println();
49     }
50
51     // Locate String objects and convert to uppercase
52     private static void convertToUppercaseStrings(List<String> list) {
53         ListIterator<String> iterator = list.listIterator();
54
55         while (iterator.hasNext()) {
56             String color = iterator.next(); // get item
57             iterator.set(color.toUpperCase()); // convert to upper case
58         }
59     }
```

Fig. 16.3 | Lists, LinkedLists and ListIterators. (Part 3 of 5.)

```
60
61     // obtain sublist and use clear method to delete sublist items
62     private static void removeItems(List<String> list,
63         int start, int end) {
64         list.subList(start, end).clear(); // remove items
65     }
66
67     // print reversed list
68     private static void printReversedList(List<String> list) {
69         ListIterator<String> iterator = list.listIterator(list.size());
70
71         System.out.printf("%nReversed List:%n");
72
73         // print list in reverse order
74         while (iterator.hasPrevious()) {
75             System.out.printf("%s ", iterator.previous());
76         }
77     }
78 }
```

Fig. 16.3 | Lists, LinkedLists and ListIterators. (Part 4 of 5.)

```
list:  
black yellow green blue violet silver gold white brown blue gray silver
```

```
list:  
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER
```

Deleting elements 4 to 6...

```
list:  
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER
```

Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK

Fig. 16.3 | Lists, LinkedLists and ListIterators. (Part 5 of 5.)

```
1 // Fig. 16.4: UsingToArray.java
2 // Viewing arrays as Lists and converting Lists to arrays.
3 import java.util.LinkedList;
4 import java.util.Arrays;
5
6 public class UsingToArray {
7     public static void main(String[] args) {
8         String[] colors = {"black", "blue", "yellow"};
9         LinkedList<String> links = new LinkedList<>(Arrays.asList(colors));
10
11         links.addLast("red"); // add as last item
12         links.add("pink"); // add to the end
13         links.add(3, "green"); // add at 3rd index
14         links.addFirst("cyan"); // add as first item
15 }
```

Fig. 16.4 | Viewing arrays as Lists and converting Lists to arrays. (Part I of 2.)

```
16     // get LinkedList elements as an array
17     colors = links.toArray(new String[links.size()]);
18
19     System.out.println("colors: ");
20
21     for (String color : colors) {
22         System.out.println(color);
23     }
24 }
25 }
```

```
colors:
cyan
black
blue
yellow
green
red
pink
```

Fig. 16.4 | Viewing arrays as Lists and converting Lists to arrays. (Part 2 of 2.)



Common Programming Error 16.2

Passing an array that contains data as `toArray`'s argument can cause logic errors. If the array's number of elements is smaller than the number of elements in the list on which `toArray` is called, a new array is allocated to store the list's elements—without preserving the array argument's elements. If the array's number of elements is greater than the number of elements in the list, the array's elements (starting at index zero) are overwritten with the list's elements. The first element of the remainder of the array is set to null to indicate the end of the list.



Software Engineering Observation 16.4

The collections framework methods are polymorphic. That is, each can operate on objects that implement specific interfaces, regardless of the underlying implementations.

Method	Description
sort	Sorts the elements of a <code>List</code> .
binarySearch	Locates an object in a <code>List</code> , using the efficient binary search algorithm which we introduced in Section 7.15 and discuss in detail in Section 19.4.
reverse	Reverses the elements of a <code>List</code> .
shuffle	Randomly orders a <code>List</code> 's elements.
fill	Sets every <code>List</code> element to refer to a specified object.
copy	Copies references from one <code>List</code> into another.

Fig. 16.5 | Some Collections methods.

Method	Description
min	Returns the smallest element in a Collection.
max	Returns the largest element in a Collection.
addAll	Appends all elements in an array to a Collection.
frequency	Calculates how many collection elements are equal to the specified element.
disjoint	Determines whether two collections have no elements in common.

Fig. 16.5 | Some Collections methods.

```
1 // Fig. 16.6: Sort1.java
2 // Collections method sort.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Sort1 {
8     public static void main(String[] args) {
9         String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
10
11         // Create and display a list containing the suits array elements
12         List<String> list = Arrays.asList(suits);
13         System.out.printf("Unsorted array elements: %s%n", list);
14
15         Collections.sort(list); // sort ArrayList
16         System.out.printf("Sorted array elements: %s%n", list);
17     }
18 }
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted array elements: [Clubs, Diamonds, Hearts, Spades]
```

Fig. 16.6 | Collections method sort.

```
1 // Fig. 16.7: Sort2.java
2 // Using a Comparator object with method sort.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Sort2 {
8     public static void main(String[] args) {
9         String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
10
11         // Create and display a list containing the suits array elements
12         List<String> list = Arrays.asList(suits); // create List
13         System.out.printf("Unsorted array elements: %s%n", list);
14
15         // sort in descending order using a comparator
16         Collections.sort(list, Collections.reverseOrder());
17         System.out.printf("Sorted list elements: %s%n", list);
18     }
19 }
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted list elements: [Spades, Hearts, Diamonds, Clubs]
```

Fig. 16.7 | Collections method sort with a Comparator object.

```
1 // Fig. 16.8: TimeComparator.java
2 // Custom Comparator class that compares two Time2 objects.
3 import java.util.Comparator;
4
5 public class TimeComparator implements Comparator<Time2> {
6     @Override
7     public int compare(Time2 time1, Time2 time2) {
8         int hourDifference = time1.getHour() - time2.getHour();
9
10        if (hourDifference != 0) { // test the hour first
11            return hourDifference;
12        }
13
14        int minuteDifference = time1.getMinute() - time2.getMinute();
15
16        if (minuteDifference != 0) { // then test the minute
17            return minuteDifference;
18        }
19
20        int secondDifference = time1.getSecond() - time2.getSecond();
21        return secondDifference;
22    }
23}
```

Fig. 16.8 | Custom Comparator class that compares two Time2 objects.

```
1 // Fig. 16.9: Sort3.java
2 // Collections method sort with a custom Comparator object.
3 import java.util.List;
4 import java.util.ArrayList;
5 import java.util.Collections;
6
7 public class Sort3 {
8     public static void main(String[] args) {
9         List<Time2> list = new ArrayList<>(); // create List
10
11         list.add(new Time2(6, 24, 34));
12         list.add(new Time2(18, 14, 58));
13         list.add(new Time2(6, 5, 34));
14         list.add(new Time2(12, 14, 58));
15         list.add(new Time2(6, 24, 22));
16     }
```

Fig. 16.9 | Collections method sort with a custom Comparator object. (Part I of 2.)

```
17     // output List elements
18     System.out.printf("Unsorted array elements:%n%s%n", list);
19
20     // sort in order using a comparator
21     Collections.sort(list, new TimeComparator());
22
23     // output List elements
24     System.out.printf("Sorted list elements:%n%s%n", list);
25 }
26 }
```

```
Unsorted array elements:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Sorted list elements:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

Fig. 16.9 | Collections method `sort` with a custom Comparator object. (Part 2 of 2.)

```
1 // Fig. 16.10: DeckOfCards.java
2 // Card shuffling and dealing with Collections method shuffle.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 // class to represent a Card in a deck of cards
8 class Card {
9     public enum Face {Ace, Deuce, Three, Four, Five, Six,
10                  Seven, Eight, Nine, Ten, Jack, Queen, King }
11     public enum Suit {Clubs, Diamonds, Hearts, Spades}
12
13     private final Face face;
14     private final Suit suit;
15
16     // constructor
17     public Card(Face face, Suit suit) {
18         this.face = face;
19         this.suit = suit;
20     }
```

Fig. 16.10 | Card shuffling and dealing with Collections method shuffle. (Part I of 5.)

```
21
22     // return face of the card
23     public Face getFace() {return face;}
24
25     // return suit of Card
26     public Suit getSuit() {return suit;}
27
28     // return String representation of Card
29     public String toString() {
30         return String.format("%s of %s", face, suit);
31     }
32 }
33
```

Fig. 16.10 | Card shuffling and dealing with Collections method `shuffle`. (Part 2 of 5.)

```
34 // class DeckOfCards declaration
35 public class DeckOfCards {
36     private List<Card> list; // declare List that will store Cards
37
38     // set up deck of Cards and shuffle
39     public DeckOfCards() {
40         Card[] deck = new Card[52];
41         int count = 0; // number of cards
42
43         // populate deck with Card objects
44         for (Card.Suit suit : Card.Suit.values()) {
45             for (Card.Face face : Card.Face.values()) {
46                 deck[count] = new Card(face, suit);
47                 ++count;
48             }
49         }
50
51         list = Arrays.asList(deck); // get List
52         Collections.shuffle(list); // shuffle deck
53     }
```

Fig. 16.10 | Card shuffling and dealing with Collections method `shuffle`. (Part 3 of 5.)

```
54
55     // output deck
56     public void printCards() {
57         // display 52 cards in four columns
58         for (int i = 0; i < list.size(); i++) {
59             System.out.printf("%-19s%s", list.get(i),
60                               ((i + 1) % 4 == 0) ? System.lineSeparator() : "");
61         }
62     }
63
64     public static void main(String[] args) {
65         DeckOfCards cards = new DeckOfCards();
66         cards.printCards();
67     }
68 }
```

Fig. 16.10 | Card shuffling and dealing with Collections method shuffle. (Part 4 of 5.)

Deuce of Clubs	Six of Spades	Nine of Diamonds	Ten of Hearts
Three of Diamonds	Five of Clubs	Deuce of Diamonds	Seven of Clubs
Three of Spades	Six of Diamonds	King of Clubs	Jack of Hearts
Ten of Spades	King of Diamonds	Eight of Spades	Six of Hearts
Nine of Clubs	Ten of Diamonds	Eight of Diamonds	Eight of Hearts
Ten of Clubs	Five of Hearts	Ace of Clubs	Deuce of Hearts
Queen of Diamonds	Ace of Diamonds	Four of Clubs	Nine of Hearts
Ace of Spades	Deuce of Spades	Ace of Hearts	Jack of Diamonds
Seven of Diamonds	Three of Hearts	Four of Spades	Four of Diamonds
Seven of Spades	King of Hearts	Seven of Hearts	Five of Diamonds
Eight of Clubs	Three of Clubs	Queen of Clubs	Queen of Spades
Six of Clubs	Nine of Spades	Four of Hearts	Jack of Clubs
Five of Spades	King of Spades	Jack of Spades	Queen of Hearts

Fig. 16.10 | Card shuffling and dealing with Collections method `shuffle`. (Part 5 of 5.)

```
1 // Fig. 16.11: Algorithms1.java
2 // Collections methods reverse, fill, copy, max and min.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Algorithms1 {
8     public static void main(String[] args) {
9         // create and display a List<Character>
10        Character[] letters = {'P', 'C', 'M'};
11        List<Character> list = Arrays.asList(letters); // get List
12        System.out.println("list contains: ");
13        output(list);
14
15        // reverse and display the List<Character>
16        Collections.reverse(list); // reverse order the elements
17        System.out.printf("%nAfter calling reverse, list contains:%n");
18        output(list);
19    }
```

Fig. 16.11 | Collections methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 1 of 4.)

```
20 // create copyList from an array of 3 Characters
21 Character[] lettersCopy = new Character[3];
22 List<Character> copyList = Arrays.asList(lettersCopy);
23
24 // copy the contents of list into copyList
25 Collections.copy(copyList, list);
26 System.out.printf("%nAfter copying, copyList contains:%n");
27 output(copyList);
28
29 // fill list with Rs
30 Collections.fill(list, 'R');
31 System.out.printf("%nAfter calling fill, list contains:%n");
32 output(list);
33 }
34 }
```

Fig. 16.11 | Collections methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 2 of 4.)

```
35 // output List information
36 private static void output(List<Character> listRef) {
37     System.out.print("The list is: ");
38
39     for (Character element : listRef) {
40         System.out.printf("%s ", element);
41     }
42
43     System.out.printf("\nMax: %s", Collections.max(listRef));
44     System.out.printf(" Min: %s\n", Collections.min(listRef));
45 }
46 }
```

Fig. 16.11 | Collections methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 3 of 4.)

```
list contains:  
The list is: P C M  
Max: P Min: C
```

```
After calling reverse, list contains:  
The list is: M C P  
Max: P Min: C
```

```
After copying, copyList contains:  
The list is: M C P  
Max: P Min: C
```

```
After calling fill, list contains:  
The list is: R R R  
Max: R Min: R
```

Fig. 16.11 | Collections methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 4 of 4.)

```
1 // Fig. 16.12: BinarySearchTest.java
2 // Collections method binarySearch.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6 import java.util.ArrayList;
7
8 public class BinarySearchTest {
9     public static void main(String[] args) {
10         // create an ArrayList<String> from the contents of colors array
11         String[] colors = {"red", "white", "blue", "black", "yellow",
12                         "purple", "tan", "pink"};
13         List<String> list = new ArrayList<>(Arrays.asList(colors));
14
15         Collections.sort(list); // sort the ArrayList
16         System.out.printf("Sorted ArrayList: %s%n", list);
17 }
```

Fig. 16.12 | Collections method binarySearch. (Part I of 3.)

```
18     // search list for various values
19     printSearchResults(list, "black");
20     printSearchResults(list, "red");
21     printSearchResults(list, "pink");
22     printSearchResults(list, "aqua"); // below lowest
23     printSearchResults(list, "gray"); // does not exist
24     printSearchResults(list, "teal"); // does not exist
25 }
26
27 // perform search and display result
28 private static void printSearchResults(
29     List<String> list, String key) {
30
31     System.out.printf("%nSearching for: %s%n", key);
32     int result = Collections.binarySearch(list, key);
33
34     if (result >= 0) {
35         System.out.printf("Found at index %d%n", result);
36     }
37     else {
38         System.out.printf("Not Found (%d)%n", result);
39     }
40 }
41 }
```

Fig. 16.12 | Collections method binarySearch. (Part 2 of 3.)

Sorted ArrayList: [black, blue, pink, purple, red, tan, white, yellow]

Searching for: black

Found at index 0

Searching for: red

Found at index 4

Searching for: pink

Found at index 2

Searching for: aqua

Not Found (-1)

Searching for: gray

Not Found (-3)

Searching for: teal

Not Found (-7)

Fig. 16.12 | Collections method binarySearch. (Part 3 of 3.)

```
1 // Fig. 16.13: Algorithms2.java
2 // Collections methods addAll, frequency and disjoint.
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Arrays;
6 import java.util.Collections;
7
8 public class Algorithms2 {
9     public static void main(String[] args) {
10         // initialize list1 and list2
11         String[] colors = {"red", "white", "yellow", "blue"};
12         List<String> list1 = Arrays.asList(colors);
13         ArrayList<String> list2 = new ArrayList<>();
14
15         list2.add("black"); // add "black" to the end of list2
16         list2.add("red"); // add "red" to the end of list2
17         list2.add("green"); // add "green" to the end of list2
18 }
```

Fig. 16.13 | Collections methods addAll, frequency and disjoint. (Part I of 3.)

```
19     System.out.print("Before addAll, list2 contains: ");
20
21     // display elements in list2
22     for (String s : list2) {
23         System.out.printf("%s ", s);
24     }
25
26     Collections.addAll(list2, colors); // add colors Strings to list2
27
28     System.out.printf("\nAfter addAll, list2 contains: ");
29
30     // display elements in list2
31     for (String s : list2) {
32         System.out.printf("%s ", s);
33     }
```

Fig. 16.13 | Collections methods addAll, frequency and disjoint. (Part 2 of 3.)

```
34
35     // get frequency of "red"
36     int frequency = Collections.frequency(list2, "red");
37     System.out.printf("%nFrequency of red in list2: %d%n", frequency);
38
39     // check whether list1 and list2 have elements in common
40     boolean disjoint = Collections.disjoint(list1, list2);
41
42     System.out.printf("list1 and list2 %s elements in common%n",
43                       (disjoint ? "do not have" : "have"));
44 }
45 }
```

```
Before addAll, list2 contains: black red green
After addAll, list2 contains: black red green red white yellow blue
Frequency of red in list2: 2
list1 and list2 have elements in common
```

Fig. 16.13 | Collections methods addAll, frequency and disjoint. (Part 3 of 3.)

```
1 // Fig. 16.14: PriorityQueueTest.java
2 // PriorityQueue test program.
3 import java.util.PriorityQueue;
4
5 public class PriorityQueueTest {
6     public static void main(String[] args) {
7         // queue of capacity 11
8         PriorityQueue<Double> queue = new PriorityQueue<>();
9
10        // insert elements to queue
11        queue.offer(3.2);
12        queue.offer(9.8);
13        queue.offer(5.4);
14    }
}
```

Fig. 16.14 | PriorityQueue test program. (Part I of 2.)

```
15    System.out.print("Polling from queue: ");
16
17    // display elements in queue
18    while (queue.size() > 0) {
19        System.out.printf("%.1f ", queue.peek()); // view top element
20        queue.poll(); // remove top element
21    }
22}
23}
```

```
Polling from queue: 3.2 5.4 9.8
```

Fig. 16.14 | PriorityQueue test program. (Part 2 of 2.)

```
1 // Fig. 16.15: SetTest.java
2 // HashSet used to remove duplicate values from array of strings.
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;
8
9 public class SetTest {
10     public static void main(String[] args) {
11         // create and display a List<String>
12         String[] colors = {"red", "white", "blue", "green", "gray",
13             "orange", "tan", "white", "cyan", "peach", "gray", "orange"};
14         List<String> list = Arrays.asList(colors);
15         System.out.printf("List: %s%n", list);
16
17         // eliminate duplicates then print the unique values
18         printNonDuplicates(list);
19     }
}
```

Fig. 16.15 | HashSet used to remove duplicate values from an array of strings. (Part I of 2.)

```
20
21     // create a Set from a Collection to eliminate duplicates
22     private static void printNonDuplicates(Collection<String> values) {
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, white, cyan, peach, gray, orange]

Nonduplicates are: tan green peach cyan red orange gray white blue

Fig. 16.15 | HashSet used to remove duplicate values from an array of strings. (Part 2 of 2.)

```
1 // Fig. 16.16: SortedSetTest.java
2 // Using SortedSets and TreeSetes.
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", "black", "tan", "grey",
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"));
```

Fig. 16.16 | Using SortedSets and TreeSetes. (Part 1 of 2.)

```
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 TreeSet. (Part 2 of 2.)



Performance Tip 16.1

The load factor in a hash table is a classic example of a memory-space/execution-time trade-off: By increasing the load factor, we get better memory utilization, but the program runs slower, due to increased hashing collisions. By decreasing the load factor, we get better program speed, because of reduced hashing collisions, but we get poorer memory utilization, because a larger portion of the hash table remains empty.

```
1 // Fig. 16.17: WordTypeCount.java
2 // Program counts the number of occurrences of each word in a String.
3 import java.util.Map;
4 import java.util.HashMap;
5 import java.util.Set;
6 import java.util.TreeSet;
7 import java.util.Scanner;
8
9 public class WordTypeCount {
10     public static void main(String[] args) {
11         // create HashMap to store String keys and Integer values
12         Map<String, Integer> myMap = new HashMap<>();
13
14         createMap(myMap); // create map based on user input
15         displayMap(myMap); // display map content
16     }
17 }
```

Fig. 16.17 | Program counts the number of occurrences of each word in a String. (Part 1 of 4.)

```
18 // create map from user input
19 private static void createMap(Map<String, Integer> map) {
20     Scanner scanner = new Scanner(System.in); // create scanner
21     System.out.println("Enter a string:"); // prompt for user input
22     String input = scanner.nextLine();
23
24     // tokenize the input
25     String[] tokens = input.split(" ");
26
27     // processing input text
28     for (String token : tokens) {
29         String word = token.toLowerCase(); // get lowercase word
30
31         // if the map contains the word
32         if (map.containsKey(word)) { // is word in map?
33             int count = map.get(word); // get current count
34             map.put(word, count + 1); // increment count
35         }
36         else {
37             map.put(word, 1); // add new word with a count of 1 to map
38         }
39     }
40 }
```

Fig. 16.17 | Program counts the number of occurrences of each word in a String. (Part 2 of 4.)

```
41
42     // display map content
43     private static void displayMap(Map<String, Integer> map) {
44         Set<String> keys = map.keySet(); // get keys
45
46         // sort keys
47         TreeSet<String> sortedKeys = new TreeSet<>(keys);
48
49         System.out.printf("%nMap contains:%nKey\t\tValue%n");
50
51         // generate output for each key in map
52         for (String key : sortedKeys) {
53             System.out.printf("%-10s%10s%n", key, map.get(key));
54         }
55
56         System.out.printf(
57             "%nsize: %d%nisEmpty: %b%n", map.size(), map.isEmpty());
58     }
59 }
```

Fig. 16.17 | Program counts the number of occurrences of each word in a String. (Part 3 of 4.)

Enter a string:

this is a sample sentence with several words this is another sample sentence with several different words

Map contains:

Key	Value
a	1
another	1
different	1
is	2
sample	2
sentence	2
several	2
this	2
with	2
words	2

size: 10

isEmpty: false

Fig. 16.17 | Program counts the number of occurrences of each word in a String. (Part 4 of 4.)



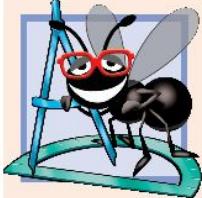
Error-Prevention Tip 16.1

Always use immutable keys with a Map. The key determines where the corresponding value is placed. If the key has changed since the insert operation, when you subsequently attempt to retrieve that value, it might not be found. In this chapter's examples, we use **Strings** as keys and **Strings** are immutable.

public static method headers

```
<T> Collection<T> synchronizedCollection(Collection<T> c)
<T> List<T> synchronizedList(List<T> aList)
<T> Set<T> synchronizedSet(Set<T> s)
<T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)
<K, V> Map<K, V> synchronizedMap(Map<K, V> m)
<K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> m)
```

Fig. 16.18 | Some synchronization wrapper methods.



Software Engineering Observation 16.5

You can use an unmodifiable wrapper to create a collection that offers read-only access to others, while allowing read–write access to yourself. You do this simply by giving others a reference to the unmodifiable wrapper while retaining for yourself a reference to the original collection.

public static method headers

```
<T> Collection<T> unmodifiableCollection(Collection<T> c)
<T> List<T> unmodifiableList(List<T> aList)
<T> Set<T> unmodifiableSet(Set<T> s)
<T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)
<K, V> Map<K, V> unmodifiableMap(Map<K, V> m)
<K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, V> m)
```

Fig. 16.19 | Some unmodifiable wrapper methods.



Common Programming Error 16.3

Calling any method that attempts to modify a collection returned by the `List`, `Set` or `Map` convenience factory methods results in an `UnsupportedOperationException`.



Software Engineering Observation 16.6

In Java, collection elements are always references to objects. The objects referenced by an immutable collection may still be mutable.

```
1 // Fig. 16.20: FactoryMethods.java
2 // Java SE 9 collection factory methods.
3 import java.util.List;
4 import java.util.Map;
5 import java.util.Set;
6
7 public class FactoryMethods {
8     public static void main(String[] args) {
9         // create a List
10        List<String> colorList = List.of("red", "orange", "yellow",
11                                         "green", "blue", "indigo", "violet");
12        System.out.printf("colorList: %s%n%n", colorList);
13
14        // create a Set
15        Set<String> colorSet = Set.of("red", "orange", "yellow",
16                                         "green", "blue", "indigo", "violet");
17        System.out.printf("colorSet: %s%n%n", colorSet);
18
19        // create a Map using method "of"
20        Map<String, Integer> dayMap = Map.of("Monday", 1, "Tuesday", 2,
21                                         "Wednesday", 3, "Thursday", 4, "Friday", 5, "Saturday", 6,
22                                         "Sunday", 7);
23        System.out.printf("dayMap: %s%n%n", dayMap);
```

Fig. 16.20 | Java SE 9 collection factory methods. (Part I of 3.)

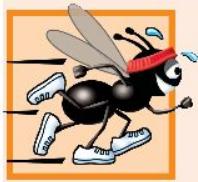
```
24
25     // create a Map using method "ofEntries" for more than 10 pairs
26     Map<String, Integer> daysPerMonthMap = Map.ofEntries(
27         Map.entry("January", 31),
28         Map.entry("February", 28),
29         Map.entry("March", 31),
30         Map.entry("April", 30),
31         Map.entry("May", 31),
32         Map.entry("June", 30),
33         Map.entry("July", 31),
34         Map.entry("August", 31),
35         Map.entry("September", 30),
36         Map.entry("October", 31),
37         Map.entry("November", 30),
38         Map.entry("December", 31)
39     );
40     System.out.printf("monthMap: %s%n", daysPerMonthMap);
41 }
42 }
```

Fig. 16.20 | Java SE 9 collection factory methods. (Part 2 of 3.)

```
colorList: [red, orange, yellow, green, blue, indigo, violet]
colorSet: [yellow, green, red, blue, violet, indigo, orange]
dayMap: {Tuesday=2, Wednesday=3, Friday=5, Thursday=4, Saturday=6, Monday=1,
Sunday=7}
monthMap: {April=30, February=28, September=30, July=31, October=31,
November=30, December=31, March=31, January=31, June=30, May=31, August=31}
```

```
colorList: [red, orange, yellow, green, blue, indigo, violet]
colorSet: [violet, yellow, orange, green, blue, red, indigo]
dayMap: {Saturday=6, Tuesday=2, Wednesday=3, Sunday=7, Monday=1, Thursday=4,
Friday=5}
monthMap: {February=28, August=31, July=31, November=30, April=30, May=31,
December=31, September=30, January=31, March=31, June=30, October=31}
```

Fig. 16.20 | Java SE 9 collection factory methods. (Part 3 of 3.)



Performance Tip 16.2

The collections returned by the convenience factory methods are optimized for up to 10 elements (for Lists and Sets) or key–value pairs (for Maps).



Software Engineering Observation 16.7

Method of is overloaded for zero to 10 elements because research showed that these handle the vast majority of cases in which immutable collections are needed.



Performance Tip 16.3

Method `of`'s overloads for zero to 10 elements eliminate the extra overhead of processing variable-length argument lists. This improves the performance of applications that create small immutable collections.



Common Programming Error 16.4

The collections returned by the convenience factory methods are not allowed to contain `null` values—these methods throw a `NullPointerException` if any argument is `null`.



Common Programming Error 16.5

Set's method of throws an `IllegalArgumentException` exception if any of its arguments are duplicates.



Common Programming Error 16.6

Map's methods `of` and `ofEntries` each throw an `IllegalArgumentException` if any of the keys are duplicates.