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 `List`s, `Set`s and `Map`s.

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 *not* contain duplicates.
`List`	An ordered collection that *can* contain duplicate elements.
`Map`	A collection that associates keys to values and *cannot* contain duplicate keys. `Map` does not derive from `Collection`.
`Queue`	Typically a *first-in, first-out* collection that models a *waiting line*; 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.


```java
// Fig. 16.2: CollectionTest.java
// Collection interface demonstrated via an ArrayList object.
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class CollectionTest {
   public static void main(String[] args) {
      // add elements in colors array to list
      String[] colors = {"MAGENTA", "RED", "WHITE", "BLUE", "CYAN"};
      List<String> list = new ArrayList<String>();

      for (String color : colors) {
         list.add(color); // adds color to end of list
      }

```

Fig. 16.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 1 of 4.)

```java
        // add elements in removeColors array to removeList
        String[] removeColors = {"RED", "WHITE", "BLUE"};
        List<String> removeList = new ArrayList<String>();

        for (String color : removeColors) {
           removeList.add(color);
        }

        // output list contents
        System.out.println("ArrayList: ");

        for (int count = 0; count < list.size(); count++) {
           System.out.printf("%s ", list.get(count));
        }
```

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


```java

        // remove from list the colors contained in removeList
        removeColors(list, removeList);

        // output list contents
        System.out.printf("%n%nArrayList after calling removeColors:%n");

        for (String color : list) {
           System.out.printf("%s ", color);
        }
     }

```

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

```java
    // remove colors specified in collection2 from collection1
    private static void removeColors(Collection<String> collection1,
       Collection<String> collection2) {
       // get iterator
       Iterator<String> iterator = collection1.iterator();

       // loop while collection has items
       while (iterator.hasNext()) {
          if (collection2.contains(iterator.next())) {
             iterator.remove(); // remove current element
          }
       }
    }
 }
```

```
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 `ArrayList`s in this example via `List` variables. This makes our code more flexible and easier to modify—if we later determine that `LinkedList`s 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 | `List`s, `LinkedList`s and `ListIterator`s. (Part 1 of 5.)

```java
        // add colors2 elements to list2
        String[] colors2 =
           {"gold", "white", "brown", "blue", "gray", "silver"};
        List<String> list2 = new LinkedList<>();

        for (String color : colors2) {
           list2.add(color);
        }

        list1.addAll(list2); // concatenate lists
        list2 = null; // release resources
        printList(list1); // print list1 elements

        convertToUppercaseStrings(list1); // convert to uppercase string
        printList(list1); // print list1 elements

        System.out.printf("%nDeleting elements 4 to 6...");
        removeItems(list1, 4, 7); // remove items 4-6 from list
        printList(list1); // print list1 elements
        printReversedList(list1); // print list in reverse order
     }
```

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


```java

     // output List contents
     private static void printList(List<String> list) {
        System.out.printf("%nlist:%n");

        for (String color : list) {
           System.out.printf("%s ", color);
        }

        System.out.println();
     }

     // locate String objects and convert to uppercase
     private static void convertToUppercaseStrings(List<String> list) {
        ListIterator<String> iterator = list.listIterator();

        while (iterator.hasNext()) {
           String color = iterator.next(); // get item
           iterator.set(color.toUpperCase()); // convert to upper case
        }
     }
```

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

```java

   // obtain sublist and use clear method to delete sublist items
   private static void removeItems(List<String> list,
      int start, int end) {
      list.subList(start, end).clear(); // remove items
   }

   // print reversed list
   private static void printReversedList(List<String> list) {
      ListIterator<String> iterator = list.listIterator(list.size());

      System.out.printf("%nReversed List:%n");

      // print list in reverse order
      while (iterator.hasPrevious()) {
         System.out.printf("%s ", iterator.previous());
      }
   }
}
```

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.)

Copyright © 2018 Pearson Education, Ltd. All Rights Reserved

```java
// Fig. 16.4: UsingToArray.java
// Viewing arrays as Lists and converting Lists to arrays.
import java.util.LinkedList;
import java.util.Arrays;

public class UsingToArray {
   public static void main(String[] args) {
      String[] colors = {"black", "blue", "yellow"};
      LinkedList<String> links = new LinkedList<>(Arrays.asList(colors));

      links.addLast("red"); // add as last item
      links.add("pink"); // add to the end
      links.add(3, "green"); // add at 3rd index
      links.addFirst("cyan"); // add as first item

```

Fig. 16.4 | Viewing arrays as Lists and converting Lists to arrays. (Part 1 of 2.)


```java
      // get LinkedList elements as an array
      colors = links.toArray(new String[links.size()]);

      System.out.println("colors: ");

      for (String color : colors) {
         System.out.println(color);
      }
   }
}
```

```
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 `List`.
`binarySearch`	Locates an object in a `List`, 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 `List`.
`shuffle`	Randomly orders a `List`'s elements.
`fill`	Sets every `List` element to refer to a specified object.
`copy`	Copies references from one `List` 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.

```java
// Fig. 16.6: Sort1.java
// Collections method sort.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort1 {
   public static void main(String[] args) {
      String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};

      // Create and display a list containing the suits array elements
      List<String> list = Arrays.asList(suits);
      System.out.printf("Unsorted array elements: %s%n", list);

      Collections.sort(list); // sort ArrayList
      System.out.printf("Sorted array elements: %s%n", list);
   }
}
```

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

Fig. 16.6 | Collections method sort.


```java
// Fig. 16.7: Sort2.java
// Using a Comparator object with method sort.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort2 {
   public static void main(String[] args) {
      String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};

      // Create and display a list containing the suits array elements
      List<String> list = Arrays.asList(suits); // create List
      System.out.printf("Unsorted array elements: %s%n", list);

      // sort in descending order using a comparator
      Collections.sort(list, Collections.reverseOrder());
      System.out.printf("Sorted list elements: %s%n", list);
   }
}
```

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

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

```java
// Fig. 16.8: TimeComparator.java
// Custom Comparator class that compares two Time2 objects.
import java.util.Comparator;

public class TimeComparator implements Comparator<Time2> {
   @Override
   public int compare(Time2 time1, Time2 time2) {
      int hourDifference = time1.getHour() - time2.getHour();

      if (hourDifference != 0) { // test the hour first
         return hourDifference;
      }

      int minuteDifference = time1.getMinute() - time2.getMinute();

      if (minuteDifference != 0) { // then test the minute
         return minuteDifference;
      }

      int secondDifference = time1.getSecond() - time2.getSecond();
      return secondDifference;
   }
}
```

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


```java
// Fig. 16.9: Sort3.java
// Collections method sort with a custom Comparator object.
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Sort3 {
   public static void main(String[] args) {
      List<Time2> list = new ArrayList<>(); // create List

      list.add(new Time2(6, 24, 34));
      list.add(new Time2(18, 14, 58));
      list.add(new Time2(6, 5, 34));
      list.add(new Time2(12, 14, 58));
      list.add(new Time2(6, 24, 22));

```

Fig. 16.9 | Collections method sort with a custom Comparator object. (Part 1 of 2.)

```
        // output List elements
        System.out.printf("Unsorted array elements:%n%s%n", list);

        // sort in order using a comparator
        Collections.sort(list, new TimeComparator());

        // output List elements
        System.out.printf("Sorted list elements:%n%s%n", list);
    }
}
```

```
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.)


```
// Fig. 16.10: DeckOfCards.java
// Card shuffling and dealing with Collections method shuffle.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

// class to represent a Card in a deck of cards
class Card {
   public enum Face {Ace, Deuce, Three, Four, Five, Six,
      Seven, Eight, Nine, Ten, Jack, Queen, King }
   public enum Suit {Clubs, Diamonds, Hearts, Spades}

   private final Face face;
   private final Suit suit;

   // constructor
   public Card(Face face, Suit suit) {
       this.face = face;
       this.suit = suit;
   }
```

Fig. 16.10 | Card shuffling and dealing with `Collections` method `shuffle`. (Part 1 of 5.)

```java

   // return face of the card
   public Face getFace() {return face;}

   // return suit of Card
   public Suit getSuit() {return suit;}

   // return String representation of Card
   public String toString() {
      return String.format("%s of %s", face, suit);
   }
}

```

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


```java
// class DeckOfCards declaration
public class DeckOfCards {
   private List<Card> list; // declare List that will store Cards

   // set up deck of Cards and shuffle
   public DeckOfCards() {
      Card[] deck = new Card[52];
      int count = 0; // number of cards

      // populate deck with Card objects
      for (Card.Suit suit : Card.Suit.values()) {
         for (Card.Face face : Card.Face.values()) {
            deck[count] = new Card(face, suit);
            ++count;
         }
      }

      list = Arrays.asList(deck); // get List
      Collections.shuffle(list);  // shuffle deck
   }
```

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

```java

    // output deck
    public void printCards() {
        // display 52 cards in four columns
        for (int i = 0; i < list.size(); i++) {
            System.out.printf("%-19s%s", list.get(i),
                ((i + 1) % 4 == 0) ? System.lineSeparator() : "");
        }
    }

    public static void main(String[] args) {
        DeckOfCards cards = new DeckOfCards();
        cards.printCards();
    }
}
```

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.)

```java
// Fig. 16.11: Algorithms1.java
// Collections methods reverse, fill, copy, max and min.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Algorithms1 {
   public static void main(String[] args) {
      // create and display a List<Character>
      Character[] letters = {'P', 'C', 'M'};
      List<Character> list = Arrays.asList(letters); // get List
      System.out.println("list contains: ");
      output(list);

      // reverse and display the List<Character>
      Collections.reverse(list); // reverse order the elements
      System.out.printf("%nAfter calling reverse, list contains:%n");
      output(list);

```

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


```java
      // create copyList from an array of 3 Characters
      Character[] lettersCopy = new Character[3];
      List<Character> copyList = Arrays.asList(lettersCopy);

      // copy the contents of list into copyList
      Collections.copy(copyList, list);
      System.out.printf("%nAfter copying, copyList contains:%n");
      output(copyList);

      // fill list with Rs
      Collections.fill(list, 'R');
      System.out.printf("%nAfter calling fill, list contains:%n");
      output(list);
   }

```

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

```java
   // output List information
   private static void output(List<Character> listRef) {
      System.out.print("The list is: ");

      for (Character element : listRef) {
         System.out.printf("%s ", element);
      }

      System.out.printf("%nMax: %s", Collections.max(listRef));
      System.out.printf("  Min: %s%n", Collections.min(listRef));
   }
}
```

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.)

```java
// Fig. 16.12: BinarySearchTest.java
// Collections method binarySearch.
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
import java.util.ArrayList;

public class BinarySearchTest {
   public static void main(String[] args) {
      // create an ArrayList<String> from the contents of colors array
      String[] colors = {"red", "white", "blue", "black", "yellow",
         "purple", "tan", "pink"};
      List<String> list = new ArrayList<>(Arrays.asList(colors));

      Collections.sort(list); // sort the ArrayList
      System.out.printf("Sorted ArrayList: %s%n", list);

```

Fig. 16.12 | Collections method binarySearch. (Part 1 of 3.)


```java
      // search list for various values
      printSearchResults(list, "black");
      printSearchResults(list, "red");
      printSearchResults(list, "pink");
      printSearchResults(list, "aqua"); // below lowest
      printSearchResults(list, "gray"); // does not exist
      printSearchResults(list, "teal"); // does not exist
   }

   // perform search and display result
   private static void printSearchResults(
      List<String> list, String key) {

      System.out.printf("%nSearching for: %s%n", key);
      int result = Collections.binarySearch(list, key);

      if (result >= 0) {
         System.out.printf("Found at index %d%n", result);
      }
      else {
         System.out.printf("Not Found (%d)%n",result);
      }
   }
}
```

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.)


```java
// Fig. 16.13: Algorithms2.java
// Collections methods addAll, frequency and disjoint.
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Algorithms2 {
   public static void main(String[] args) {
      // initialize list1 and list2
      String[] colors = {"red", "white", "yellow", "blue"};
      List<String> list1 = Arrays.asList(colors);
      ArrayList<String> list2 = new ArrayList<>();

      list2.add("black"); // add "black" to the end of list2
      list2.add("red"); // add "red" to the end of list2
      list2.add("green"); // add "green" to the end of list2

```

Fig. 16.13 | Collections methods addAll, frequency and disjoint. (Part 1 of 3.)

```java
        System.out.print("Before addAll, list2 contains: ");

        // display elements in list2
        for (String s : list2) {
           System.out.printf("%s ", s);
        }

        Collections.addAll(list2, colors); // add colors Strings to list2

        System.out.printf("%nAfter addAll, list2 contains: ");

        // display elements in list2
        for (String s : list2) {
           System.out.printf("%s ", s);
        }
```

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


```java

        // get frequency of "red"
        int frequency = Collections.frequency(list2, "red");
        System.out.printf("%nFrequency of red in list2: %d%n", frequency);

        // check whether list1 and list2 have elements in common
        boolean disjoint = Collections.disjoint(list1, list2);

        System.out.printf("list1 and list2 %s elements in common%n",
           (disjoint ? "do not have" : "have"));
     }
  }
```

```
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.)

```java
// Fig. 16.14: PriorityQueueTest.java
// PriorityQueue test program.
import java.util.PriorityQueue;

public class PriorityQueueTest {
   public static void main(String[] args) {
      // queue of capacity 11
      PriorityQueue<Double> queue = new PriorityQueue<>();

      // insert elements to queue
      queue.offer(3.2);
      queue.offer(9.8);
      queue.offer(5.4);

```

Fig. 16.14 | PriorityQueue test program. (Part 1 of 2.)


```java
      System.out.print("Polling from queue: ");

      // display elements in queue
      while (queue.size() > 0) {
         System.out.printf("%.1f ", queue.peek()); // view top element
         queue.poll(); // remove top element
      }
   }
}
```

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

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

```java
// Fig. 16.15: SetTest.java
// HashSet used to remove duplicate values from array of strings.
import java.util.List;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.Collection;

public class SetTest {
   public static void main(String[] args) {
      // create and display a List<String>
      String[] colors = {"red", "white", "blue", "green", "gray",
         "orange", "tan", "white", "cyan", "peach", "gray", "orange"};
      List<String> list = Arrays.asList(colors);
      System.out.printf("List: %s%n", list);

      // eliminate duplicates then print the unique values
      printNonDuplicates(list);
   }
```

Fig. 16.15 | HashSet used to remove duplicate values from an array of strings. (Part 1 of 2.)


```java

   // create a Set from a Collection to eliminate duplicates
   private static void printNonDuplicates(Collection<String> values) {
      // create a HashSet
      Set<String> set = new HashSet<>(values);

      System.out.printf("%nNonduplicates are: ");

      for (String value : set) {
         System.out.printf("%s ", value);
      }

      System.out.println();
   }
}
```

```
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.)

```java
// Fig. 16.16: SortedSetTest.java
// Using SortedSets and TreeSets.
import java.util.Arrays;
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetTest {
   public static void main(String[] args) {
      // create TreeSet from array colors
      String[] colors = {"yellow", "green", "black", "tan", "grey",
         "white", "orange", "red", "green"};
      SortedSet<String> tree = new TreeSet<>(Arrays.asList(colors));

      System.out.print("sorted set: ");
      printSet(tree);

      // get headSet based on "orange"
      System.out.print("headSet (\"orange\"):  ");
      printSet(tree.headSet("orange"));

      // get tailSet based upon "orange"
      System.out.print("tailSet (\"orange\"):  ");
      printSet(tree.tailSet("orange"));
```

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


```java

      // get first and last elements
      System.out.printf("first: %s%n", tree.first());
      System.out.printf("last : %s%n", tree.last());
   }

   // output SortedSet using enhanced for statement
   private static void printSet(SortedSet<String> set) {
      for (String s : set) {
         System.out.printf("%s ", s);
      }

      System.out.println();
   }
}
```

```
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. (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.


```java
// Fig. 16.17: WordTypeCount.java
// Program counts the number of occurrences of each word in a String.
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

public class WordTypeCount {
   public static void main(String[] args) {
      // create HashMap to store String keys and Integer values
      Map<String, Integer> myMap = new HashMap<>();

      createMap(myMap); // create map based on user input
      displayMap(myMap); // display map content
   }

```

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

```java
    // create map from user input
    private static void createMap(Map<String, Integer> map) {
       Scanner scanner = new Scanner(System.in); // create scanner
       System.out.println("Enter a string:"); // prompt for user input
       String input = scanner.nextLine();

       // tokenize the input
       String[] tokens = input.split(" ");

       // processing input text
       for (String token : tokens) {
          String word = token.toLowerCase(); // get lowercase word

          // if the map contains the word
          if (map.containsKey(word)) { // is word in map?
             int count = map.get(word); // get current count
             map.put(word, count + 1); // increment count
          }
          else {
             map.put(word, 1); // add new word with a count of 1 to map
          }
       }
    }
```

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


```java

    // display map content
    private static void displayMap(Map<String, Integer> map) {
       Set<String> keys = map.keySet(); // get keys

       // sort keys
       TreeSet<String> sortedKeys = new TreeSet<>(keys);

       System.out.printf("%nMap contains:%nKey\t\tValue%n");

       // generate output for each key in map
       for (String key : sortedKeys) {
          System.out.printf("%-10s%10s%n", key, map.get(key));
       }

       System.out.printf(
          "%nsize: %d%nisEmpty: %b%n", map.size(), map.isEmpty());
    }
 }
```

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 `String`s as keys and `String`s 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.


```java
// Fig. 16.20: FactoryMethods.java
// Java SE 9 collection factory methods.
import java.util.List;
import java.util.Map;
import java.util.Set;

public class FactoryMethods {
   public static void main(String[] args) {
      // create a List
      List<String> colorList = List.of("red", "orange", "yellow",
         "green", "blue", "indigo", "violet");
      System.out.printf("colorList: %s%n%n", colorList);

      // create a Set
      Set<String> colorSet = Set.of("red", "orange", "yellow",
         "green", "blue", "indigo", "violet");
      System.out.printf("colorSet: %s%n%n", colorSet);

      // create a Map using method "of"
      Map<String, Integer> dayMap = Map.of("Monday", 1, "Tuesday", 2,
         "Wednesday", 3, "Thursday", 4, "Friday", 5, "Saturday", 6,
         "Sunday", 7);
      System.out.printf("dayMap: %s%n%n", dayMap);
```

Fig. 16.20 | Java SE 9 collection factory methods. (Part 1 of 3.)

```java

        // create a Map using method "ofEntries" for more than 10 pairs
        Map<String, Integer> daysPerMonthMap = Map.ofEntries(
           Map.entry("January", 31),
           Map.entry("February", 28),
           Map.entry("March", 31),
           Map.entry("April", 30),
           Map.entry("May", 31),
           Map.entry("June", 30),
           Map.entry("July", 31),
           Map.entry("August", 31),
           Map.entry("September", 30),
           Map.entry("October", 31),
           Map.entry("November", 30),
           Map.entry("December", 31)
        );
        System.out.printf("monthMap: %s%n", daysPerMonthMap);
    }
}
```

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 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.