



Chapter 16

Generic Collections

Java How to Program, 10/e



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.
- Use prebuilt generic data structures from the collections framework.
- Use iterators to “walk through” a collection.
- Use persistent hash tables manipulated with objects of class **Properties**.
- Learn about synchronization and modifiability wrappers.



-
- 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** Stack Class of Package `java.util`
 - 16.9** Class `PriorityQueue` and Interface `Queue`
 - 16.10** Sets
-



16.11 Maps

16.12 Properties Class

16.13 Synchronized Collections

16.14 Unmodifiable Collections

16.15 Abstract Implementations

16.16 Wrap-Up



16.1 Introduction

▶ What is a Framework?

- A framework is a set of classes and interfaces which provide a ready-made architecture.
- Java **collections framework**
 - Contains *prebuilt* generic data structures



16.2 Collections Overview

- ▶ A **collection** is a data structure—actually, an object—that can hold references to other objects.
 - Usually, collections contain references to objects of any type that has the *is-a* relationship with the type stored in the collection.
- ▶ Figure 16.1 lists some of the collections framework interfaces.
- ▶ Package `java.util`.

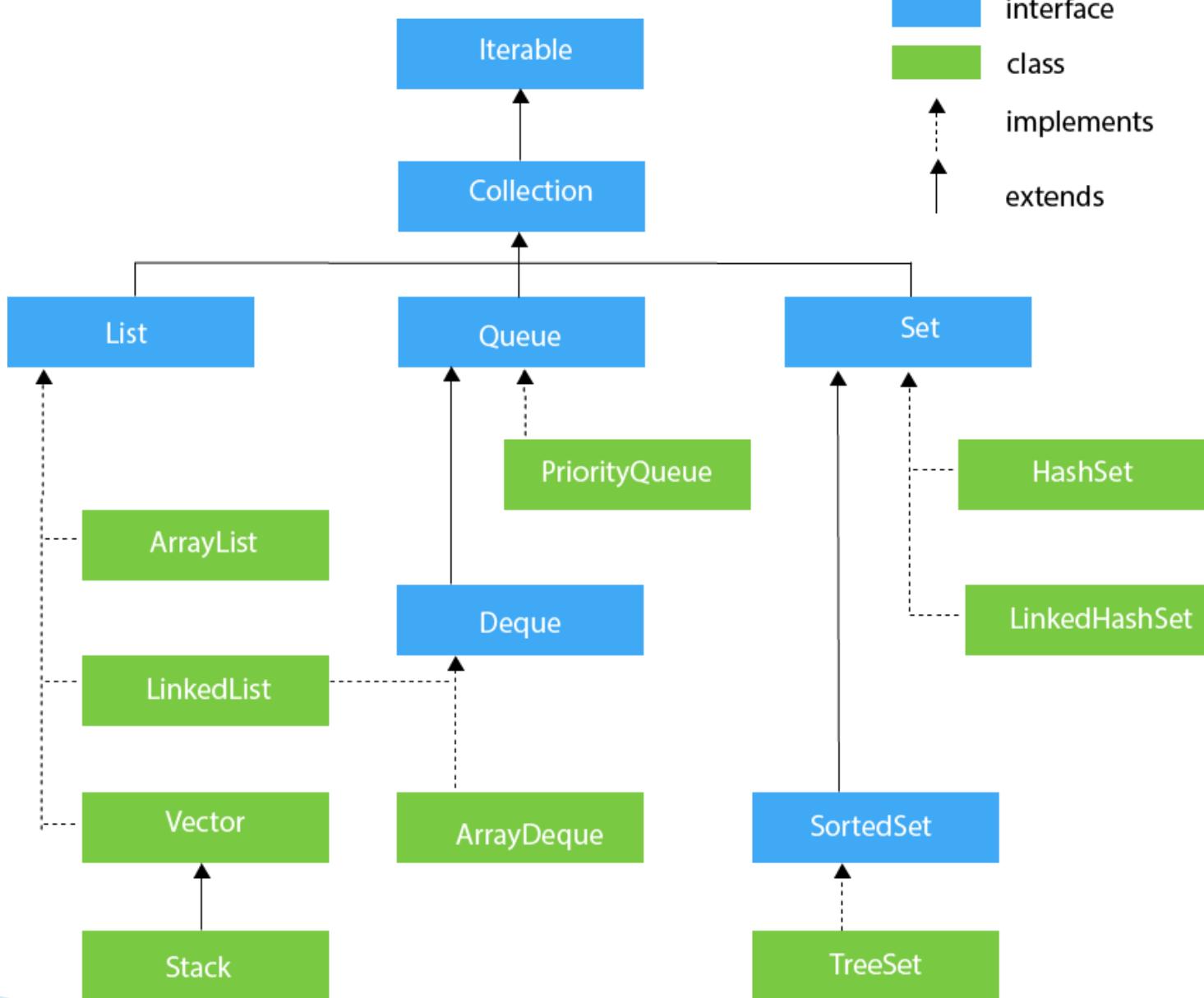


- ▶ A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:
 - **Interfaces**
 - **Implementations**
 - **Algorithms**



Using Collections

- ▶ `import java.util.*`
or `import java.util.Collection;`
- ▶ There is a sister class, `java.util.Collections`; that provides a number of algorithms for use with collections: `sort`, `binarySearch`, `copy`, `shuffle`, `reverse`, `max`, `min`, etc.





Collection interface

- ▶ <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Collection.html>
- ▶ Don't confuse with Collections class



List Interface

- ▶ A List is an ordered Collection Lists may contain duplicate elements.
- ▶ In addition to the operations inherited from Collection, the List interface includes operations for the following:
 - Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.
 - Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.
 - Iteration — extends Iterator semantics to take advantage of the list's sequential nature. The listIterator methods provide this behavior.
 - Range-view — The sublist method performs arbitrary range operations on the list.



List Interface

- ▶ This list interface is implemented by various classes like ArrayList, Vector, Stack, etc. Since all the subclasses implement the list, we can instantiate a list object with any of these classes. For example,

- ▶ `List <T> al = new ArrayList<>();`
- ▶ `List <T> ll = new LinkedList<>();`
- ▶ `List <T> v = new Vector<>();`
- ▶ Where T is the type of the object



- ▶ Type Inference with the `<>` Notation
 - Java SE 7 introduced *type inferencing* with the `<>` notation—known as the **diamond notation**—in statements that declare and create generic type variables and objects. For example:

```
List<String> list = new ArrayList<>();
```

- Java uses the type in angle brackets on the left of the declaration (that is, `String`) as the type stored in the `ArrayList` created on the right side of the declaration.



ArrayList

- ▶ ArrayList provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.
- ▶ The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- ▶ Java ArrayList allows us to randomly access the list.



- ▶ ArrayList can not be used for primitive types, like int, char, etc. We will need a wrapper class for such cases.



16.3 Type-Wrapper Classes

- ▶ Each primitive type has a corresponding **type-wrapper class** (in package `java.lang`).
 - `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` and `Short`.
- ▶ Each type-wrapper class enables you to manipulate primitive-type values as objects.
- ▶ **Collections cannot manipulate variables of primitive types.**
 - They can manipulate objects of the type-wrapper classes, because every class ultimately derives from `Object`.



16.3 Type-Wrapper Classes (cont.)

- ▶ Each of the numeric type-wrapper classes—`Byte`, `Short`, `Integer`, `Long`, `Float` and `Double`—extends class `Number`.
- ▶ The type-wrapper classes are `final` classes, so you cannot extend them.
- ▶ Primitive types do not have methods, so the methods related to a primitive type are located in the corresponding type-wrapper class.



16.4 Autoboxing and Auto-Unboxing

- ▶ A **boxing conversion** converts a value of a primitive type to an object of the corresponding type-wrapper class.
- ▶ An **unboxing conversion** converts an object of a type-wrapper class to a value of the corresponding primitive type.
- ▶ These conversions are performed automatically—called **autoboxing** and **auto-unboxing**.
- ▶ Example:
 - ```
// create integerArray
Integer[] integerArray = new Integer[5];

// assign Integer 10 to integerArray[0]
integerArray[0] = 10;

// get int value of Integer
int value = integerArray[0];
```



## 16.5 Interface Collection and Class Collections

- ▶ Interface **Collection** contains **bulk operations** for *adding*, *clearing* and *comparing* objects in a collection.
- ▶ A **Collection** can be converted to an array.
- ▶ Interface **Collection** provides a method that returns an **Iterator** object, which allows a program to walk through the collection and remove elements from the collection during the iteration.
- ▶ Class **Collections** provides **static** methods that *search*, *sort* and perform other operations on collections.



## 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.*



## 16.6 Lists

- ▶ In addition to the methods inherited from **Collection**, **List** provides methods for manipulating elements via their indices, manipulating a specified range of elements, searching for elements and obtaining a **ListIterator** to access the elements.
- ▶ Interface **List** is implemented by several classes, including **ArrayList**, **LinkedList** and **Vector**.
- ▶ Autoboxing occurs when you add primitive-type values to objects of these classes, because they store only references to objects.



## 16.6 Lists (cont.)

- ▶ Class **ArrayList** and **Vector** are resizable-array implementations of **List**.
- ▶ Inserting an element between existing elements of an **ArrayList** or **Vector** is an *inefficient* operation.
- ▶ A **LinkedList** enables *efficient* insertion (or removal) of elements in the middle of a collection, but is much less efficient than an **ArrayList** for jumping to a specific element in the collection.
- ▶ The primary difference between **ArrayList** and **Vector** is that operations on **Vectors** are *synchronized* by default, whereas those on **ArrayLists** are not.
- ▶ *Unsynchronized collections provide better performance than synchronized ones.*
- ▶ For this reason, **ArrayList** is typically preferred over **Vector** in programs that do not share a collection among threads.



## Performance Tip 16.1

*ArrayLists behave like Vectors without synchronization and therefore execute faster than Vectors, because ArrayLists do not have the overhead of thread synchronization.*



## 16.6.1 ArrayList and Iterator

- ▶ List method `add` adds an item to the end of a list.
- ▶ List method `size` returns the number of elements.
- ▶ List method `get` retrieves an individual element's value from the specified index.
- ▶ Collection method `iterator` gets an `Iterator` for a `Collection`.
- ▶ Iterator- method `hasNext` determines whether there are more elements to iterate through.
  - Returns `true` if another element exists and `false` otherwise.
- ▶ Iterator method `next` obtains a reference to the next element.
- ▶ Collection method `contains` determine whether a `Collection` contains a specified element.
- ▶ Iterator method `remove` removes the current element from a `Collection`.



# Iteration options

- ▶ There are three ways to traverse collections:
  - (1) using aggregate operations (will be discussed later)
  - ▶ (2) with the for-each construct and
  - ▶ (3) by using Iterators.



# for-each

- ▶ The for-each construct allows you to concisely traverse a collection or array using a for loop. The following code uses the for-each construct to print out each element of a collection on a separate line.

```
for (Object o : collection)
 System.out.println(o);
```



# Iterators

- ▶ You get an Iterator for a collection by calling its iterator method. The following is the Iterator interface.

```
public interface Iterator<E> {
 boolean hasNext();
 E next();
 void remove(); //optional
}
```



# Iterator

- ▶ The following method shows you how to use an Iterator to filter an arbitrary Collection — that is, traverse the collection removing specific elements.

```
void filter(Collection<?> c) {
 for (Iterator<?> it = c.iterator();
it.hasNext();)
 if (!cond(it.next()))
 it.remove();
}
```



# Iterator

- ▶ The `hasNext` method returns true if the iteration has more elements, and the `next` method returns the next element in the iteration.
- ▶ The `remove` method removes the last element that was returned by `next` from the underlying Collection.
- ▶ The `remove` method may be called only once per call to `next` and throws an exception if this rule is violated.



# Iterator

- ▶ Note that `Iterator.remove` is the only safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.
- ▶ Use `Iterator` instead of the `for-each` construct when you need to:
  - Remove the current element. The `for-each` construct hides the iterator, so you cannot call `remove`. Therefore, the `for-each` construct is not usable for filtering.



## 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 immediate 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.*



# Collection Interface Bulk Operations

- ▶ Bulk operations perform an operation on an entire Collection.

- `containsAll` — returns `true` if the target `Collection` contains all of the elements in the specified `Collection`.
- `addAll` — adds all of the elements in the specified `Collection` to the target `Collection`.
- `removeAll` — removes from the target `Collection` all of its elements that are also contained in the specified `Collection`.
- `retainAll` — removes from the target `Collection` all its elements that are *not* also contained in the specified `Collection`.  
those elements in the target `Collection` that are also contained in the specified `Collection`.
- `clear` — removes all elements from the `Collection`.



# Collection Interface Array Operations

- ▶ The `toArray` methods are provided as a bridge between collections and older APIs that expect arrays on input.
- ▶ The array operations allow the contents of a Collection to be translated into an array.



# Collection Interface Array Operations

- ▶ For example, suppose that c is a Collection. To dump the contents of c into a newly allocated array of Object
  - `Object[] a = c.toArray();`
- ▶ Suppose that c is known to contain only strings, dump the contents of c into a newly allocated array of String whose length is identical to the number of elements in c.

```
String[] a = c.toArray(new String[0]);
```



```
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 {
10 public static void main(String[] args)
11 {
12 // add elements in colors array to list
13 String[] colors = {"MAGENTA", "RED", "WHITE", "BLUE", "CYAN"};
14 List<String> list = new ArrayList<String>();
15
16 for (String color : colors)
17 list.add(color); // adds color to end of list
18
19 // add elements in removeColors array to removeList
20 String[] removeColors = {"RED", "WHITE", "BLUE"};
21 List<String> removeList = new ArrayList<String>();
22
23 for (String color : removeColors)
24 removeList.add(color);
```

---

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



---

```
25 // output list contents
26 System.out.println("ArrayList: ");
27
28 for (int count = 0; count < list.size(); count++)
29 System.out.printf("%s ", list.get(count));
30
31 // remove from list the colors contained in removeList
32 removeColors(list, removeList);
33
34 // output list contents
35 System.out.printf("%n%nArrayList after calling removeColors:%n");
36
37 for (String color : list)
38 System.out.printf("%s ", color);
39
40 }
41
```

---

**Fig. 16.2** | Collection interface demonstrated via an `ArrayList` object. (Part 2 of 3.)



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

ArrayList:

MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:

MAGENTA CYAN

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



## 16.6.2 LinkedList

- ▶ List method `addAll` appends all elements of a collection to the end of a `List`.
- ▶ List method `listIterator` gets A `List`'s bidirectional iterator.
- ▶ List-Iterator method `set` replaces the current element to which the iterator refers with the specified object.
- ▶ List method `subList` obtains a portion of a `List`.
  - This is a so-called `range-view method`, which enables the program to view a portion of the list.



## 16.6.2 LinkedList (cont.)

- ▶ **List** method `clear` remove the elements of a **List**.
- ▶ **List** method `size` returns the number of items in the **List**.
- ▶ **ListIterator** method `hasPrevious` determines whether there are more elements while traversing the list backward.
- ▶ **ListIterator** method `previous` gets the previous element from the list.



```
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 {
9 public static void main(String[] args)
10 {
11 // add colors elements to list1
12 String[] colors =
13 {"black", "yellow", "green", "blue", "violet", "silver"};
14 List<String> list1 = new LinkedList<>();
15
16 for (String color : colors)
17 list1.add(color);
18
19 // add colors2 elements to list2
20 String[] colors2 =
21 {"gold", "white", "brown", "blue", "gray", "silver"};
22 List<String> list2 = new LinkedList<>();
23 }
}
```

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



```
24 for (String color : colors2)
25 list2.add(color);
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 }
39
```

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



```
40 // output List contents
41 private static void printList(List<String> list)
42 {
43 System.out.printf("%nlist:%n");
44
45 for (String color : list)
46 System.out.printf("%s ", color);
47
48 System.out.println();
49 }
50
51 // Locate String objects and convert to uppercase
52 private static void convertToUppercaseStrings(List<String> list)
53 {
54 ListIterator<String> iterator = list.listIterator();
55
56 while (iterator.hasNext())
57 {
58 String color = iterator.next(); // get item
59 iterator.set(color.toUpperCase()); // convert to upper case
60 }
61 }
62 }
```

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



```
63 // obtain sublist and use clear method to delete sublist items
64 private static void removeItems(List<String> list,
65 int start, int end)
66 {
67 list.subList(start, end).clear(); // remove items
68 }
69
70 // print reversed list
71 private static void printReversedList(List<String> list)
72 {
73 ListIterator<String> iterator = list.listIterator(list.size());
74
75 System.out.printf("%nReversed List:%n");
76
77 // print list in reverse order
78 while (iterator.hasPrevious())
79 System.out.printf("%s ", iterator.previous());
80 }
81 } // end class ListTest
```

**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.)



## 16.6.2 LinkedList (cont.)

- ▶ Class **Arrays** provides **static** method **asList** to view an array as a **List** collection.
  - A **List** view allows you to manipulate the array as if it were a list.
  - This is useful for adding the elements in an array to a collection and for sorting array elements.
- ▶ Any modifications made through the **List** view change the array, and any modifications made to the array change the **List** view.
- ▶ The only operation permitted on the view returned by **asList** is **set**, which changes the value of the view and the backing array.
  - Any other attempts to change the view result in an **UnsupportedOperationException**.
- ▶ **List** method **toArray** gets an array from a **List** collection.



```
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 {
8 // creates a LinkedList, adds elements and converts to array
9 public static void main(String[] args)
10 {
11 String[] colors = {"black", "blue", "yellow"};
12 LinkedList<String> links = new LinkedList<>(Arrays.asList(colors));
13
14 links.addLast("red"); // add as last item
15 links.add("pink"); // add to the end
16 links.add(3, "green"); // add at 3rd index
17 links.addFirst("cyan"); // add as first item
18
19 // get LinkedList elements as an array
20 colors = links.toArray(new String[links.size()]);
21 }
```

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



```
22 System.out.println("colors: ");
23
24 for (String color : colors)
25 System.out.println(color);
26 }
27 } // end class UsingToArray
```

```
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 to `toArray` can cause logic errors. If the number of elements in the array 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 number of elements in the array is greater than the number of elements in the list, the elements of the array (starting at index zero) are overwritten with the list's elements. Array elements that are not overwritten retain their values.*



## 16.6.2 `LinkedList` (cont.)

- ▶ `LinkedList` method `addLast` adds an element to the end of a `List`.
- ▶ `LinkedList` method `add` also adds an element to the end of a `List`.
- ▶ `LinkedList` method `addFirst` adds an element to the beginning of a `List`.

## 16.7 Collections Methods

- ▶ Class **Collections** provides several high-performance algorithms for manipulating collection elements.
- ▶ The algorithms (Fig. 16.5) are implemented as **static** methods.



| Method                    | Description                                                                                                                                                              |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sort</code>         | Sorts the elements of a <code>List</code> .                                                                                                                              |
| <code>binarySearch</code> | Locates an object in a <code>List</code> , using the high-performance binary search algorithm which we introduced in Section 7.15 and discuss in detail in Section 19.4. |
| <code>reverse</code>      | Reverses the elements of a <code>List</code> .                                                                                                                           |
| <code>shuffle</code>      | Randomly orders a <code>List</code> 's elements.                                                                                                                         |
| <code>fill</code>         | Sets every <code>List</code> element to refer to a specified object.                                                                                                     |
| <code>copy</code>         | Copies references from one <code>List</code> into another.                                                                                                               |
| <code>min</code>          | Returns the smallest element in a <code>Collection</code> .                                                                                                              |
| <code>max</code>          | Returns the largest element in a <code>Collection</code> .                                                                                                               |
| <code>addAll</code>       | Appends all elements in an array to a <code>Collection</code> .                                                                                                          |
| <code>frequency</code>    | Calculates how many collection elements are equal to the specified element.                                                                                              |
| <code>disjoint</code>     | Determines whether two collections have no elements in common.                                                                                                           |

**Fig. 16.5** | Collections methods.



## 16.7.1 Method sort

- ▶ **Method sort** sorts the elements of a `List`
  - The elements must implement the `Comparable` interface.
  - The order is determined by the natural order of the elements' type as implemented by a `compareTo` method.
  - Method `compareTo` is declared in interface `Comparable` and is sometimes called the `natural comparison method`.
  - The `sort` call may specify as a second argument a `Comparator` object that determines an alternative ordering of the elements.



```
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 {
9 public static void main(String[] args)
10 {
11 String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
12
13 // Create and display a list containing the suits array elements
14 List<String> list = Arrays.asList(suits);
15 System.out.printf("Unsorted array elements: %s%n", list);
16
17 Collections.sort(list); // sort ArrayList
18 System.out.printf("Sorted array elements: %s%n", list);
19 }
20 } // end class Sort1
```

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

**Fig. 16.6** | Collections method sort.



## 16.7.1 Method sort (cont.)

- ▶ The **Comparator** interface is used for sorting a **Collection**'s elements in a different order.
- ▶ The **static Collections method reverseOrder** returns a **Comparator** object that orders the collection's elements in reverse order.



```
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 {
9 public static void main(String[] args)
10 {
11 String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
12
13 // Create and display a list containing the suits array elements
14 List<String> list = Arrays.asList(suits); // create List
15 System.out.printf("Unsorted array elements: %s%n", list);
16
17 // sort in descending order using a comparator
18 Collections.sort(list, Collections.reverseOrder());
19 System.out.printf("Sorted list elements: %s%n", list);
20 }
21 } // end class Sort2
```

**Fig. 16.7** | Collections method sort with a Comparator object. (Part 1 of 2.)



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

Sorted list elements: [Spades, Hearts, Diamonds, Clubs]

**Fig. 16.7** | Collections method sort with a Comparator object. (Part 2 of 2.)



## 16.7.1 Method sort (cont.)

- ▶ Figure 16.8 creates a custom **Comparator** class, named **TimeComparator**, that implements interface **Comparator** to compare two **Time2** objects.
- ▶ Class **Time2**, declared in Fig. 8.5, represents times with hours, minutes and seconds.
- ▶ Class **TimeComparator** implements interface **Comparator**, a generic type that takes one type argument.
- ▶ A class that implements **Comparator** must declare a **compare** method that receives two arguments and returns a negative integer if the first argument is less than the second, 0 if the arguments are equal or a positive integer if the first argument is greater than the second.



```
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 {
7 @Override
8 public int compare(Time2 time1, Time2 time2)
9 {
10 int hourDifference = time1.getHour() - time2.getHour();
11
12 if (hourDifference != 0) // test the hour first
13 return hourCompare;
14
15 int minuteDifference = time1.getMinute() - time2.getMinute();
16
17 if (minuteDifference != 0) // then test the minute
18 return minuteDifference;
19
20 int secondDifference = time1.getSecond() - time2.getSecond();
21 return secondDifference;
22 }
23 } // end class TimeComparator
```

**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 {
9 public static void main(String[] args)
10 {
11 List<Time2> list = new ArrayList<>(); // create List
12
13 list.add(new Time2(6, 24, 34));
14 list.add(new Time2(18, 14, 58));
15 list.add(new Time2(6, 05, 34));
16 list.add(new Time2(12, 14, 58));
17 list.add(new Time2(6, 24, 22));
18
19 // output List elements
20 System.out.printf("Unsorted array elements:%n%s%n", list);
21 }
}
```

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



```
22 // sort in order using a comparator
23 Collections.sort(list, new TimeComparator());
24
25 // output List elements
26 System.out.printf("Sorted list elements:%n%s%n", list);
27 }
28 } // end class Sort3
```

```
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.)



## 16.7.2 Method `shuffle`

- ▶ Method `shuffle` randomly orders a `List`'s elements.



---

```
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 {
10 public static enum Face {Ace, Deuce, Three, Four, Five, Six,
11 Seven, Eight, Nine, Ten, Jack, Queen, King };
12 public static enum Suit {Clubs, Diamonds, Hearts, Spades};
13
14 private final Face face;
15 private final Suit suit;
16
17 // constructor
18 public Card(Face face, Suit suit)
19 {
20 this.face = face;
21 this.suit = suit;
22 }
23 }
```

---

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



```
24 // return face of the card
25 public Face getFace()
26 {
27 return face;
28 }
29
30 // return suit of Card
31 public Suit getSuit()
32 {
33 return suit;
34 }
35
36 // return String representation of Card
37 public String toString()
38 {
39 return String.format("%s of %s", face, suit);
40 }
41 } // end class Card
42
```

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



```
43 // class DeckOfCards declaration
44 public class DeckOfCards
45 {
46 private List<Card> list; // declare List that will store Cards
47
48 // set up deck of Cards and shuffle
49 public DeckOfCards()
50 {
51 Card[] deck = new Card[52];
52 int count = 0; // number of cards
53
54 // populate deck with Card objects
55 for (Card.Suit suit: Card.Suit.values())
56 {
57 for (Card.Face face: Card.Face.values())
58 {
59 deck[count] = new Card(face, suit);
60 ++count;
61 }
62 }
63 }
```

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



```
64 list = Arrays.asList(deck); // get List
65 Collections.shuffle(list); // shuffle deck
66 } // end DeckOfCards constructor
67
68 // output deck
69 public void printCards()
70 {
71 // display 52 cards in two columns
72 for (int i = 0; i < list.size(); i++)
73 System.out.printf("%-19s%s", list.get(i),
74 ((i + 1) % 4 == 0) ? "%n" : "");
75 }
76
77 public static void main(String[] args)
78 {
79 DeckOfCards cards = new DeckOfCards();
80 cards.printCards();
81 }
82 } // end class DeckOfCards
```

**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.)



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

- ▶ **Collections method `reverse`** reverses the order of the elements in a `List`
- ▶ **Method `fill`** overwrites elements in a `List` with a specified value.
- ▶ **Method `copy`** takes two arguments—a destination `List` and a source `List`.
  - Each source `List` element is copied to the destination `List`.
  - The destination `List` must be at least as long as the source `List`; otherwise, an `IndexOutOfBoundsException` occurs.
  - If the destination `List` is longer, the elements not overwritten are unchanged.
- ▶ Methods `min` and `max` each operate on any `Collection`.
  - Method `min` returns the smallest element in a `Collection`, and method `max` returns the largest element in a `Collection`.



```
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 {
9 public static void main(String[] args)
10 {
11 // create and display a List<Character>
12 Character[] letters = {'P', 'C', 'M'};
13 List<Character> list = Arrays.asList(letters); // get List
14 System.out.println("list contains: ");
15 output(list);
16
17 // reverse and display the List<Character>
18 Collections.reverse(list); // reverse order the elements
19 System.out.printf("%nAfter calling reverse, list contains:%n");
20 output(list);
21 }
```

---

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



```
22 // create copyList from an array of 3 Characters
23 Character[] lettersCopy = new Character[3];
24 List<Character> copyList = Arrays.asList(lettersCopy);
25
26 // copy the contents of list into copyList
27 Collections.copy(copyList, list);
28 System.out.printf("%nAfter copying, copyList contains:%n");
29 output(copyList);
30
31 // fill list with Rs
32 Collections.fill(list, 'R');
33 System.out.printf("%nAfter calling fill, list contains:%n");
34 output(list);
35 }
36
37 // output List information
38 private static void output(List<Character> listRef)
39 {
40 System.out.print("The list is: ");
41
42 for (Character element : listRef)
43 System.out.printf("%s ", element);
44 }
```

**Fig. 16.11** | Collections methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 2 of 3.)



```
45 System.out.printf("%nMax: %s", Collections.max(listRef));
46 System.out.printf(" Min: %s%n", Collections.min(listRef));
47 }
48 } // end class Algorithms1
```

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 3 of 3.)



## 16.7.4 Method `binarySearch`

- ▶ `static Collections method binarySearch` locates an object in a `List`.
  - If the object is found, its index is returned.
  - If the object is not found, `binarySearch` returns a negative value.



```
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 {
10 public static void main(String[] args)
11 {
12 // create an ArrayList<String> from the contents of colors array
13 String[] colors = {"red", "white", "blue", "black", "yellow",
14 "purple", "tan", "pink"};
15 List<String> list =
16 new ArrayList<>(Arrays.asList(colors));
17
18 Collections.sort(list); // sort the ArrayList
19 System.out.printf("Sorted ArrayList: %s%n", list);
20 }
```

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



```
21 // search list for various values
22 printSearchResults(list, "black"); // first item
23 printSearchResults(list, "red"); // middle item
24 printSearchResults(list, "pink"); // last item
25 printSearchResults(list, "aqua"); // below lowest
26 printSearchResults(list, "gray"); // does not exist
27 printSearchResults(list, "teal"); // does not exist
28 }
29
30 // perform search and display result
31 private static void printSearchResults(
32 List<String> list, String key)
33 {
34 int result = 0;
35
36 System.out.printf("%nSearching for: %s%n", key);
37 result = Collections.binarySearch(list, key);
38
39 if (result >= 0)
40 System.out.printf("Found at index %d%n", result);
41 else
42 System.out.printf("Not Found (%d)%n", result);
43 }
44 } // end class BinarySearchTest
```

**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.)



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

- ▶ Collections method `addAll` takes two arguments—a **Collection** into which to *insert* the new element(s) and an array that provides elements to be inserted.
- ▶ Collections method `frequency` takes two arguments—a **Collection** to be searched and an **Object** to be searched for in the collection.
  - Method `frequency` returns the number of times that the second argument appears in the collection.
- ▶ Collections method `disjoint` takes two **Collections** and returns `true` if they have *no elements in common*.



```
1 // Fig. 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 {
10 public static void main(String[] args)
11 {
12 // initialize list1 and list2
13 String[] colors = {"red", "white", "yellow", "blue"};
14 List<String> list1 = Arrays.asList(colors);
15 ArrayList<String> list2 = new ArrayList<>();
16
17 list2.add("black"); // add "black" to the end of list2
18 list2.add("red"); // add "red" to the end of list2
19 list2.add("green"); // add "green" to the end of list2
20
21 System.out.print("Before addAll, list2 contains: ");
22 }
```

---

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



```
23 // display elements in list2
24 for (String s : list2)
25 System.out.printf("%s ", s);
26
27 Collections.addAll(list2, colors); // add colors Strings to list2
28
29 System.out.printf("%nAfter addAll, list2 contains: ");
30
31 // display elements in list2
32 for (String s : list2)
33 System.out.printf("%s ", s);
34
35 // get frequency of "red"
36 int frequency = Collections.frequency(list2, "red");
37 System.out.printf(
38 "%nFrequency of red in list2: %d%n", frequency);
39
40 // check whether list1 and list2 have elements in common
41 boolean disjoint = Collections.disjoint(list1, list2);
42
43 System.out.printf("list1 and list2 %s elements in common%n",
44 (disjoint ? "do not have" : "have"));
45 }
46 } // end class Algorithms2
```

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



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.)



## 16.8 Stack Class of Package `java.util`

- ▶ Class `Stack` in the Java utilities package (`java-.util`) extends class `Vector` to implement a stack data structure.
- ▶ `Stack` method `push` adds a `Number` object to the top of the stack.
- ▶ `Stack` method `pop` removes the top element of the stack.
  - If there are no elements in the `Stack`, method `pop` throws an `EmptyStackException`, which terminates the loop.
- ▶ Method `peek` returns the top element of the stack without popping the element off the stack.
- ▶ Method `isEmpty` determines whether the stack is empty.



```
1 // Fig. 16.14: StackTest.java
2 // Stack class of package java.util.
3 import java.util.Stack;
4 import java.util.EmptyStackException;
5
6 public class StackTest
7 {
8 public static void main(String[] args)
9 {
10 Stack<Number> stack = new Stack<>(); // create a Stack
11
12 // use push method
13 stack.push(12L); // push long value 12L
14 System.out.println("Pushed 12L");
15 printStack(stack);
16 stack.push(34567); // push int value 34567
17 System.out.println("Pushed 34567");
18 printStack(stack);
19 stack.push(1.0F); // push float value 1.0F
20 System.out.println("Pushed 1.0F");
21 printStack(stack);
22 stack.push(1234.5678); // push double value 1234.5678
23 System.out.println("Pushed 1234.5678 ");
24 printStack(stack);
```

**Fig. 16.14** | Stack class of package java.util. (Part 1 of 4.)



---

```
25
26 // remove items from stack
27 try
28 {
29 Number removedObject = null;
30
31 // pop elements from stack
32 while (true)
33 {
34 removedObject = stack.pop(); // use pop method
35 System.out.printf("Popped %s%n", removedObject);
36 printStack(stack);
37 }
38 }
39 catch (EmptyStackException emptyStackException)
40 {
41 emptyStackException.printStackTrace();
42 }
43 }
44 }
```

---

**Fig. 16.14** | Stack class of package java.util. (Part 2 of 4.)



---

```
45 // display Stack contents
46 private static void printStack(Stack<Number> stack)
47 {
48 if (stack.isEmpty())
49 System.out.printf("stack is empty%n%n"); // the stack is empty
50 else // stack is not empty
51 System.out.printf("stack contains: %s (top)%n", stack);
52 }
53 } // end class StackTest
```

---

**Fig. 16.14** | Stack class of package java.util. (Part 3 of 4.)



```
Pushed 12L
stack contains: [12] (top)
Pushed 34567
stack contains: [12, 34567] (top)
Pushed 1.0F
stack contains: [12, 34567, 1.0] (top)
Pushed 1234.5678
stack contains: [12, 34567, 1.0, 1234.5678] (top)
Popped 1234.5678
stack contains: [12, 34567, 1.0] (top)
Popped 1.0
stack contains: [12, 34567] (top)
Popped 34567
stack contains: [12] (top)
Popped 12
stack is empty

java.util.EmptyStackException
 at java.util.Stack.peek(Unknown Source)
 at java.util.Stack.pop(Unknown Source)
 at StackTest.main(StackTest.java:34)
```

**Fig. 16.14** | Stack class of package java.util. (Part 4 of 4.)



## Error-Prevention Tip 16.1

*Because Stack extends Vector, all public Vector methods can be called on Stack objects, even if the methods do not represent conventional stack operations. For example, Vector method add can be used to insert an element anywhere in a stack—an operation that could “corrupt” the stack. When manipulating a Stack, only methods push and pop should be used to add elements to and remove elements from the Stack, respectively. In Section 21.5, we create a Stack class using composition so that the Stack provides in its public interface only the capabilities that should be allowed by a Stack.*



## 16.9 Class PriorityQueue and Interface Queue

- ▶ Interface **Queue** extends interface **Collection** and provides additional operations for inserting, removing and inspecting elements in a queue.
- ▶ **PriorityQueue** orders elements by their natural ordering.
  - Elements are inserted in priority order such that the highest-priority element (i.e., the largest value) will be the first element removed from the **PriorityQueue**.
- ▶ Common **PriorityQueue** operations are
  - **offer** to insert an element at the appropriate location based on priority order
  - **poll** to remove the highest-priority element of the priority queue
  - **peek** to get a reference to the highest-priority element of the priority queue
  - **clear** to remove all elements in the priority queue
  - **size** to get the number of elements in the queue.



```
1 // Fig. 16.15: PriorityQueueTest.java
2 // PriorityQueue test program.
3 import java.util.PriorityQueue;
4
5 public class PriorityQueueTest
6 {
7 public static void main(String[] args)
8 {
9 // queue of capacity 11
10 PriorityQueue<Double> queue = new PriorityQueue<>();
11
12 // insert elements to queue
13 queue.offer(3.2);
14 queue.offer(9.8);
15 queue.offer(5.4);
16
17 System.out.print("Polling from queue: ");
18 }
}
```

**Fig. 16.15** | PriorityQueue test program. (Part 1 of 2.)



```
19 // display elements in queue
20 while (queue.size() > 0)
21 {
22 System.out.printf("%.1f ", queue.peek()); // view top element
23 queue.poll(); // remove top element
24 }
25 }
26 } // end class PriorityQueueTest
```

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

**Fig. 16.15** | PriorityQueue test program. (Part 2 of 2.)



## 16.10 Sets

- ▶ A **Set** is an *unordered Collection* of unique elements (i.e., *no duplicates*).
- ▶ The collections framework contains several **Set** implementations, including **HashSet** and **TreeSet**.
- ▶ **HashSet** stores its elements in a *hash table*, and **TreeSet** stores its elements in a *tree*.



```
1 // Fig. 16.16: 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 {
11 public static void main(String[] args)
12 {
13 // create and display a List<String>
14 String[] colors = {"red", "white", "blue", "green", "gray",
15 "orange", "tan", "white", "cyan", "peach", "gray", "orange"};
16 List<String> list = Arrays.asList(colors);
17 System.out.printf("List: %s%n", list);
18
19 // eliminate duplicates then print the unique values
20 printNonDuplicates(list);
21 }
22 }
```

**Fig. 16.16** | HashSet used to remove duplicate values from an array of strings. (Part 1 of 2.)



```
23 // create a Set from a Collection to eliminate duplicates
24 private static void printNonDuplicates(Collection<String> values)
25 {
26 // create a HashSet
27 Set<String> set = new HashSet<>(values);
28
29 System.out.printf("%nNonduplicates are: ");
30
31 for (String value : set)
32 System.out.printf("%s ", value);
33
34 System.out.println();
35 }
36 } // end class SetTest
```

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

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

**Fig. 16.16** | HashSet used to remove duplicate values from an array of strings. (Part 2 of 2.)



## 16.10 Sets (cont.)

- ▶ The collections framework also includes the **SortedSet interface** (which extends **Set**) for sets that maintain their elements in *sorted* order.
- ▶ Class **TreeSet** implements **SortedSet**.
- ▶ **TreeSet** method **headSet** gets a subset of the **TreeSet** in which every element is less than the specified value.
- ▶ **TreeSet** method **tailSet** gets a subset in which each element is greater than or equal to the specified value.
- ▶ **SortedSet** methods **first** and **last** get the smallest and largest elements of the set, respectively.



```
1 // Fig. 16.17: 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 {
9 public static void main(String[] args)
10 {
11 // create TreeSet from array colors
12 String[] colors = {"yellow", "green", "black", "tan", "grey",
13 "white", "orange", "red", "green"};
14 SortedSet<String> tree = new TreeSet<>(Arrays.asList(colors));
15
16 System.out.print("sorted set: ");
17 printSet(tree);
18
19 // get headSet based on "orange"
20 System.out.print("headSet (\"orange\"): ");
21 printSet(tree.headSet("orange"));
22 }
}
```

**Fig. 16.17** | Using SortedSets and TreeSets. (Part 1 of 3.)



```
23 // get tailSet based upon "orange"
24 System.out.print("tailSet (\"orange\"): ");
25 printSet(tree.tailSet("orange"));
26
27 // get first and last elements
28 System.out.printf("first: %s%n", tree.first());
29 System.out.printf("last : %s%n", tree.last());
30 }
31
32 // output SortedSet using enhanced for statement
33 private static void printSet(SortedSet<String> set)
34 {
35 for (String s : set)
36 System.out.printf("%s ", s);
37
38 System.out.println();
39 }
40 } // end class SortedSetTest
```

**Fig. 16.17** | Using SortedSets and TreeSet. (Part 2 of 3.)



```
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. 16.17** | Using SortedSets and TreeSet. (Part 3 of 3.)



## 16.11 Maps

- ▶ **Maps** associate keys to values.
  - The keys in a **Map** must be unique, but the associated values need not be.
  - If a **Map** contains both unique keys and unique values, it is said to implement a **one-to-one mapping**.
  - If only the keys are unique, the **Map** is said to implement a **many-to-one mapping**—many keys can map to one value.
- ▶ Three of the several classes that implement interface **Map** are **Hashtable**, **HashMap** and **TreeMap**.
- ▶ **Hashtables** and **HashMaps** store elements in hash tables, and **TreeMaps** store elements in trees.



## 16.11 Maps (Cont.)

- ▶ Interface **SortedMap** extends **Map** and maintains its keys in *sorted* order—either the elements' *natural* order or an order specified by a **Comparator**.
- ▶ Class **TreeMap** implements **SortedMap**.
- ▶ Hashing is a high-speed scheme for converting keys into unique array indices.
- ▶ A hash table's **load factor** affects the performance of hashing schemes.
  - The load factor is the ratio of the number of occupied cells in the hash table to the total number of cells in the hash table.
- ▶ The closer this ratio gets to 1.0, the greater the chance of collisions.



```
1 // Fig. 16.18: 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 {
11 public static void main(String[] args)
12 {
13 // create HashMap to store String keys and Integer values
14 Map<String, Integer> myMap = new HashMap<>();
15
16 createMap(myMap); // create map based on user input
17 displayMap(myMap); // display map content
18 }
19 }
```

**Fig. 16.18** | Program counts the number of occurrences of each word in a String.  
(Part I of 5.)



---

```
20 // create map from user input
21 private static void createMap(Map<String, Integer> map)
22 {
23 Scanner scanner = new Scanner(System.in); // create scanner
24 System.out.println("Enter a string:"); // prompt for user input
25 String input = scanner.nextLine();
26
27 // tokenize the input
28 String[] tokens = input.split(" ");
29
30 // processing input text
31 for (String token : tokens)
32 {
33 String word = token.toLowerCase(); // get lowercase word
34 }
```

---

**Fig. 16.18** | Program counts the number of occurrences of each word in a *String*.  
(Part 2 of 5.)



```
35 // if the map contains the word
36 if (map.containsKey(word)) // is word in map
37 {
38 int count = map.get(word); // get current count
39 map.put(word, count + 1); // increment count
40 }
41 else
42 map.put(word, 1); // add new word with a count of 1 to map
43 }
44 }
45
46 // display map content
47 private static void displayMap(Map<String, Integer> map)
48 {
49 Set<String> keys = map.keySet(); // get keys
50
51 // sort keys
52 TreeSet<String> sortedKeys = new TreeSet<>(keys);
53
54 System.out.printf("%nMap contains:%nKey\t\tValue%n");
55 }
```

**Fig. 16.18** | Program counts the number of occurrences of each word in a String.  
(Part 3 of 5.)



```
56 // generate output for each key in map
57 for (String key : sortedKeys)
58 System.out.printf("%-10s%10s%n", key, map.get(key));
59
60 System.out.printf(
61 "%nsize: %d%isEmpty: %b%n", map.size(), map.isEmpty());
62 }
63 } // end class WordTypeCount
```

**Fig. 16.18** | Program counts the number of occurrences of each word in a *String*.  
(Part 4 of 5.)



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. 16.18** | Program counts the number of occurrences of each word in a String.

(Part 5 of 5.)



## 16.11 Maps (Cont.)

- ▶ Map method `containsKey` determines whether a key is in a map.
- ▶ Map method `put` creates a new entry or replaces an existing entry's value.
  - Method `put` returns the key's prior associated value, or `null` if the key was not in the map.
- ▶ Map method `get` obtain the specified key's associated value in the map.
- ▶ HashMap method `keySet` returns a set of the keys.
- ▶ Map method `size` returns the number of key/value pairs in the Map.
- ▶ Map method `isEmpty` returns a boolean indicating whether the Map is empty.



## Error-Prevention Tip 16.2

*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.*



## 16.13 Synchronized Collections

- ▶ **Synchronization wrappers** are used for collections that might be accessed by multiple threads.
- ▶ A **wrapper** object receives method calls, adds thread synchronization and delegates the calls to the wrapped collection object.
- ▶ The **Collections API** provides a set of **static** methods for wrapping collections as synchronized versions.
- ▶ Method headers for the synchronization wrappers are listed in Fig. 16.20.



## 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. 16.20** | Synchronization wrapper methods.



## 16.14 Unmodifiable Collections

- ▶ The `Collections` class provides a set of `static` methods that create `unmodifiable wrappers` for collections.
- ▶ Unmodifiable wrappers throw `UnsupportedOperationExceptions` if attempts are made to modify the collection.
- ▶ In an unmodifiable collection, the references stored in the collection are not modifiable, but the objects they refer *are modifiable* unless they belong to an immutable class like `String`.
- ▶ Headers for these methods are listed in Fig. 16.21.



## Software Engineering Observation 16.6

*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. 16.21** | Unmodifiable wrapper methods.



## 16.15 Abstract Implementations

- ▶ The collections framework provides various abstract implementations of **Collection** interfaces from which you can quickly “flesh out” complete customized implementations.
- ▶ These include
  - a thin **Collection** implementation called an **AbstractCollection**
  - a **List** implementation that allows *array-like access* to its elements called an **AbstractList**
  - a **Map** implementation called an **AbstractMap**
  - a **List** implementation that allows *sequential access* (from beginning to end) to its elements called an **AbstractSequentialList**
  - a **Set** implementation called an **AbstractSet**
  - a **Queue** implementation called **AbstractQueue**.