# Chapter 16
# Generic Collections

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

# 16.1 Introduction

- Java collections framework
  - prebuilt data structures
  - interfaces and methods for manipulating those 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 that are all of the same type.
- Package `java.util`.

| 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. |
| `Queue` | Typically a first-in, first-out collection that models a waiting line; other orders can be specified. |

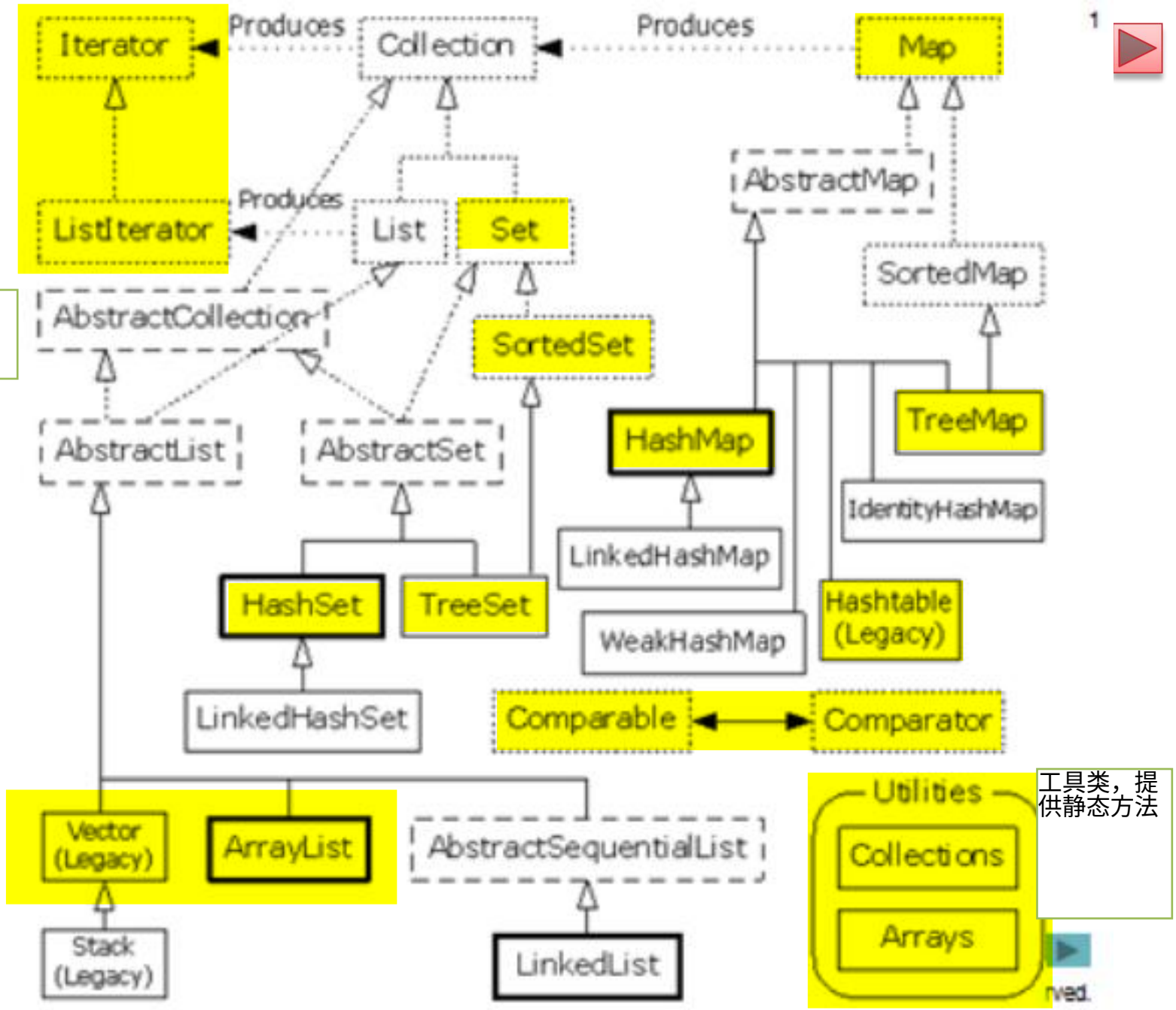
collection的子类

过滤重复元素

**Fig. 20.1** | Some collections-framework interfaces.

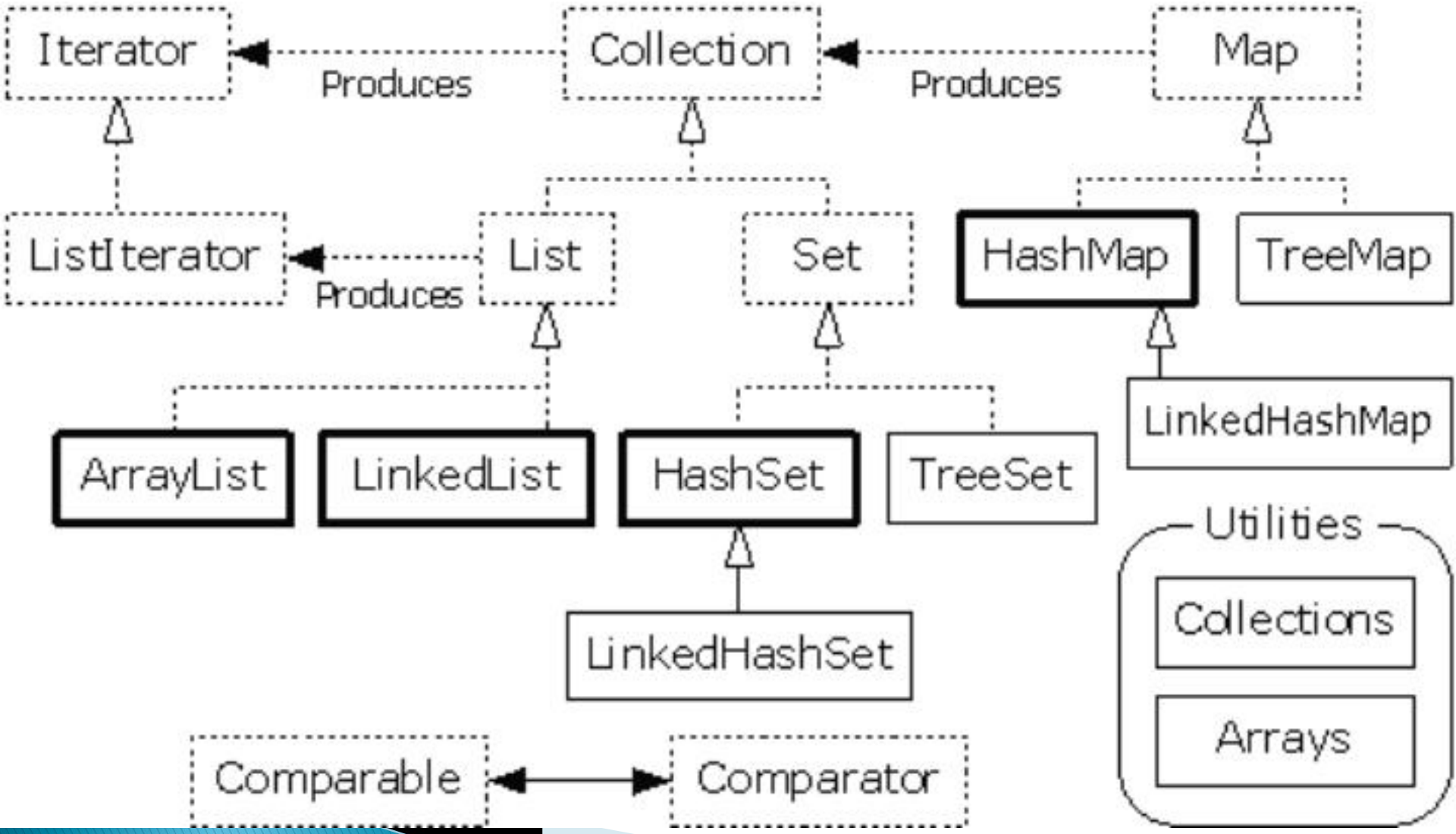
# Collections framework

密集小点：接口
实线：类
虚线：抽象类




工具类，提供静态方法

# Simplified Collections framework



# 16.3 Type-Wrapper Classes for Primitive Types

- Each primitive type has a corresponding type-wrapper class (in package `java.lang`).
  - Boolean, Byte, Character, Double, Float, Integer, Long and Short.
- The type-wrapper classes:
  - extends class `Number`
  - are `final` classes, so you cannot extend them.

不能被继承　类final
不能被重写　方法final
不能被改变　变量final

- Collections cannot manipulate variables of primitive types.

# 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 can be 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 20.1

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

## Software Engineering Observation 20.2

*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.*

# 16.6 Lists

- A `List` (sometimes called a sequence) is a `Collection` that can contain duplicate elements.
- `List` indices are zero based.
- `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`.

多线程：vector替换arraylist

- The primary difference between `ArrayList` and `Vector` is that `Vector`s are synchronized by default, whereas `ArrayList`s are not.
- Unsynchronized collections provide better performance than synchronized ones.
- Inserting an element between existing elements of an `ArrayList` or `Vector` is an inefficient operation.
- `LinkedList` s efficient insertion (or removal) of elements in the middle n.

## Software Engineering Observation 20.3

`LinkedLists` *can be used to create stacks, queues and deques (double-ended queues, pronounced "decks"). The collections framework provides implementations of some of these data structures.*

# 16.6.1 ArrayList and Iterator

- List method add adds an item to the end of a list.
- List method size retursn 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 a `Collection` contains more elements. 确定一个集合是否包含更多的元素
  - 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`.

```
1  // Fig. 20.2: 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
```

**Fig. 20.2** | `Collection` interface demonstrated via an `ArrayList` object. (Part 1 of 3.)

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

**Fig. 20.2** | `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 Color
54         } // end while
55      } // end method removeColors
56   } // end class CollectionTest
```

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN
```

**Fig. 20.2** | `Collection` interface demonstrated via an `ArrayList` object. (Part 3 of 3.)

## Common Programming Error 20.1

*If a collection is modified by one of its methods after an iterator is created for that collection, the iterator immediately becomes invalid—operations performed with the iterator after this point throw* `ConcurrentModificationExceptions`. *For this reason, iterators are said to be "fail fast."*

# 16.6.1 ArrayList and Iterator

- New in Java SE 7: Type Inference with the <> Notation
  - Lines 14 and 21 specify the type stored in the `ArrayList` (that is, `String`) on the left and right sides of the initialization statements.
  - Java SE 7 supports type inferencing with the <> notation in statements that declare and create generic type variables and objects. For example, line 14 can be written as:

```
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.

# 16.6.2 LinkedList

- List method addAll appends all elements of a collecton to the end of a `List`.
- List method listIterator gets A `List`'s bidirectional iterator.
- String method toUpperCase gets an uppercase version of a `String`.
- ListIterator method set replaces the current element to which the iterator refers with the specified object.
- String method toLowerCase returns a lowercase version of a `String`.
- 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.

# 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. 20.3: 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< String >();
15
16       for ( String color : colors )
17          list1.add( color );
18
```

**Fig. 20.3** | Lists, LinkedLists and ListIterators. (Part 1 of 5.)

```
19          // add colors2 elements to list2
20          String[] colors2 =
21             { "gold", "white", "brown", "blue", "gray", "silver" };
22          List< String > list2 = new LinkedList< String >();
23
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.print( "\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       } // end main
39
```

collection
添加所有

**Fig. 20.3** | Lists, LinkedLists and ListIterators. (Part 2 of 5.)

```
40      // output List contents
41      private static void printList( List< String > list )
42      {
43         System.out.println( "\nlist: " );
44
45         for ( String color : list )
46            System.out.printf( "%s ", color );
47
48         System.out.println();
49      } // end method printList
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         } // end while
61      } // end method convertToUppercaseStrings
62
```

**Fig. 20.3** | 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      } // end method removeItems
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.println( "\nReversed List:" );
76
77         // print list in reverse order
78         while ( iterator.hasPrevious() )
79            System.out.printf( "%s ", iterator.previous() );
80      } // end method printReversedList
81   } // end class ListTest
```

**Fig. 20.3** | 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. 20.3** | `Lists`, `LinkedLists` and `ListIterators`. (Part 5 of 5.)

```
1  // Fig. 20.4: 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  {
```

**Fig. 20.4** | Viewing arrays as `Lists` and converting `Lists` to arrays. (Part 1 of 3.)

```
 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
13         LinkedList< String > links =
14            new LinkedList< String >( Arrays.asList( colors ) );
15
16         links.addLast( "red" ); // add as last item
17         links.add( "pink" ); // add to the end
18         links.add( 3, "green" ); // add at 3rd index
19         links.addFirst( "cyan" ); // add as first item
20
21         // get LinkedList elements as an array
22         colors = links.toArray( new String[ links.size() ] );
23
24         System.out.println( "colors: " );
25
26         for ( String color : colors )
27            System.out.println( color );
28      } // end main
29   } // end class UsingToArray
```

看作list，可以做list的处理

**Fig. 20.4** | Viewing arrays as `List`s and converting `List`s to arrays. (Part 2 of 3.)

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

**Fig. 20.4** | Viewing arrays as `Lists` and converting `Lists` to arrays. (Part 3 of 3.)

# 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`.

## Common Programming Error 20.2

*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.*

# 16.7 Collections Methods

- Class `Collections` provides several high-performance algorithms for manipulating collection elements.

| Method | Description |
|---|---|
| `sort` | Sorts the elements of a `List`. |
| `binarySearch` | Locates an object in a `List`. |
| `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. |
| `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. 20.5** | `Collections` methods.

## Software Engineering Observation 20.4

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

# 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. 20.6: 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 ); // create List
15       System.out.printf( "Unsorted array elements: %s\n", list );
16
17       Collections.sort( list ); // sort ArrayList
18
19       // output list
20       System.out.printf( "Sorted array elements: %s\n", list );
21    } // end main
22 } // end class Sort1
```

**Fig. 20.6** | `Collections` method `sort`. (Part 1 of 2.)

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

**Fig. 20.6** | `Collections` method `sort`. (Part 2 of 2.)

# 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. 20.7: 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
20       // output List elements
21       System.out.printf( "Sorted list elements: %s\n", list );
22    } // end main
23 } // end class Sort2
```

**Fig. 20.7** | `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. 20.7** | `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. 20.8: 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     public int compare( Time2 time1, Time2 time2 )
8     {
9        int hourCompare = time1.getHour() - time2.getHour(); // compare hour
10
```

**Fig. 20.8** | Custom `Comparator` class that compares two `Time2` objects. (Part 1 of 2.)

```
11          // test the hour first
12          if ( hourCompare != 0 )
13             return hourCompare;
14
15          int minuteCompare =
16             time1.getMinute() - time2.getMinute(); // compare minute
17
18          // then test the minute
19          if ( minuteCompare != 0 )
20             return minuteCompare;
21
22          int secondCompare =
23             time1.getSecond() - time2.getSecond(); // compare second
24
25          return secondCompare; // return result of comparing seconds
26       } // end method compare
27    } // end class TimeComparator
```

**Fig. 20.8** | Custom `Comparator` class that compares two `Time2` objects. (Part 2 of 2.)

```
1  // Fig. 20.9: 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< Time2 >(); // 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
```

**Fig. 20.9** | `Collections` method `sort` with a custom `Comparator` object. (Part 1 of 2.)

```
19         // output List elements
20         System.out.printf( "Unsorted array elements:\n%s\n", list );
21
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      } // end main
28   } // end class Sort3
```

自己定义一个新的comparator

```
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. 20.9** | `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. 20.10: 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; // face of card
15    private final Suit suit; // suit of card
16
17    // two-argument constructor
18    public Card( Face cardFace, Suit cardSuit )
19    {
20        face = cardFace; // initialize face of card
21        suit = cardSuit; // initialize suit of card
22    } // end two-argument Card constructor
23
```

**Fig. 20.10** | 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      } // end method getFace
29
30      // return suit of Card
31      public Suit getSuit()
32      {
33         return suit;
34      } // end method getSuit
35
36      // return String representation of Card
37      public String toString()
38      {
39         return String.format( "%s of %s", face, suit );
40      } // end method toString
41   } // end class Card
42
```

**Fig. 20.10** | 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           } // end for
62        } // end for
63
```

**Fig. 20.10** | 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       } // end method printCards
76
77       public static void main( String[] args )
78       {
79          DeckOfCards cards = new DeckOfCards();
80          cards.printCards();
81       } // end main
82    } // end class DeckOfCards
```

**Fig. 20.10** | 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. 20.10** | 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. 20.11: 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.println( "\nAfter calling reverse, list contains: " );
20       output( list );
21
```

**Fig. 20.11** | `Collections` methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 1 of 4.)

```
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.println( "\nAfter copying, copyList contains: " );
29          output( copyList );
30
31          // fill list with Rs
32          Collections.fill( list, 'R' );
33          System.out.println( "\nAfter calling fill, list contains: " );
34          output( list );
35       } // end main
36
```

**Fig. 20.11** | `Collections` methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 2 of 4.)

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

**Fig. 20.11** | `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. 20.11** | `Collections` methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 4 of 4.)

# 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. 20.12: 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< String >( Arrays.asList( colors ) );
17
18       Collections.sort( list ); // sort the ArrayList
19       System.out.printf( "Sorted ArrayList: %s\n", list );
20
```

**Fig. 20.12** | `Collections` method `binarySearch`. (Part 1 of 3.)

```
21          // search list for various values
22          printSearchResults( list, colors[ 3 ] ); // first item
23          printSearchResults( list, colors[ 0 ] ); // middle item
24          printSearchResults( list, colors[ 7 ] ); // last item
25          printSearchResults( list, "aqua" ); // below lowest
26          printSearchResults( list, "gray" ); // does not exist
27          printSearchResults( list, "teal" ); // does not exist
28       } // end main
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       } // end method printSearchResults
44    } // end class BinarySearchTest
```

**Fig. 20.12** | 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. 20.12** | `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 `Collection`s and returns `true` if they have no elements in common.

```
1  // Fig. 20.13: 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< String >();
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. 20.13** | `Collections` methods `addAll`, `frequency` and `disjoint`. (Part 1 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.print( "\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       } // end main
```

**Fig. 20.13** | `Collections` methods `addAll`, `frequency` and `disjoint`. (Part 2 of 3.)

```
46   } // end class Algorithms2
```

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

# 16.8 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. 20.15: 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< Double >();
```

**Fig. 20.15** | `PriorityQueue` test program. (Part 1 of 2.)

```
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
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          } // end while
25       } // end main
26    } // end class PriorityQueueTest
```

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

**Fig. 20.15** | `PriorityQueue` test program. (Part 2 of 2.)

# 16.9 Sets

- A Set is an unordered `Collection` of unique elements (i.e., no duplicate elements).
- 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. 20.16: 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    } // end main
22
```

**Fig. 20.16** | `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< String >( values );
28
29         System.out.print( "\nNonduplicates are: " );
30
31         for ( String value : set )
32            System.out.printf( "%s ", value );
33
34         System.out.println();
35      } // end method printNonDuplicates
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. 20.16** | `HashSet` used to remove duplicate values from an array of strings. (Part 2 of 2.)

# 16.9 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. 20.17: 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 =
15          new TreeSet< String >( Arrays.asList( colors ) );
16
17       System.out.print( "sorted set: " );
18       printSet( tree ); // output contents of tree
19
20       // get headSet based on "orange"
21       System.out.print( "headSet (\"orange\"):  " );
22       printSet( tree.headSet( "orange" ) );
23
```

**Fig. 20.17** | Using SortedSets and TreeSets. (Part 1 of 3.)

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

**Fig. 20.17** | Using `SortedSet`s and `TreeSet`s. (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. 20.17** | Using `SortedSet`s and `TreeSet`s. (Part 3 of 3.)

# 16.10 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.
- `Hashtable`s and `HashMap`s store elements in hash tables, and `TreeMap`s store elements in trees.

# 16.10 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.

## Performance Tip 20.2

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

```
1  // Fig. 20.18: 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< String, Integer >();
15
16       createMap( myMap ); // create map based on user input
17       displayMap( myMap ); // display map content
18    } // end main
19
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a `String`. (Part 1 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
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a `String`. (Part 2 of 5.)

```
30          // processing input text
31          for ( String token : tokens )
32          {
33             String word = token.toLowerCase(); // get lowercase word
34
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             } // end if
41             else
42                map.put( word, 1 ); // add new word with a count of 1 to map
43          } // end for
44       } // end method createMap
45
```

**Fig. 20.18** | Program counts the number of occurrences of each word in a `String`. (Part 3 of 5.)

```
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< String >( keys );
53
54         System.out.println( "\nMap contains:\nKey\t\tValue" );
55
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\nisEmpty: %b\n", map.size(), map.isEmpty() );
62      } // end method displayMap
63   } // end class WordTypeCount
```

**Fig. 20.18** | 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. 20.18** | Program counts the number of occurrences of each word in a `String`. (Part 5 of 5.)

# 16.10 Maps (Cont.)

- Map method containsKey determines whether a key is in a map.
- Map method put creates a new entry in the map 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.

# 16.11 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.

**`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. 20.20** | Synchronization wrapper methods.

# 16.12 Unmodifiable Collections

- The `Collections` class provides a set of `static` methods that create unmodifiable wrappers for collections.
- Unmodifiable wrappers throw `UnsupportedOperationException`s 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

**`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. 20.21** | Unmodifiable wrapper methods.

## Software Engineering Observation 20.5

*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.*

# 16.13 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 random access to its elements called an AbstractList
  - a `Map` implementation called an AbstractMap
  - a `List` implementation that allows sequential access to its elements called an AbstractSequentialList
  - a `Set` implementation called an AbstractSet
  - a `Queue` implementation called AbstractQueue.

# 16.14 Java SE 9: Convenience Factory Methods for Immutable Collections

- Java SE 9 adds new static convenience factory methods to interfaces List, Set and Map that enable you to create small immutable collections—they cannot be modified once they are created
- The convenience factory methods return custom collection objects that are truly immutable and optimized to store small collections.

# 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`.

```java
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);

      // 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);
   }
}
```

# List.of(.....)

| | | |
|---|---|---|
| static <E> List<E> | of() | Returns an immutable list containing zero elements. |
| static <E> List<E> | of(E e1) | Returns an immutable list containing one element. |
| static <E> List<E> | of(E... elements) | Returns an immutable list containing an arbitrary number of element |
| static <E> List<E> | of(E e1, E e2) | Returns an immutable list containing two elements. |
| static <E> List<E> | of(E e1, E e2, E e3) | Returns an immutable list containing three elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4) | Returns an immutable list containing four elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5) | Returns an immutable list containing five elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5, E e6) | Returns an immutable list containing six elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7) | Returns an immutable list containing seven elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8) | Returns an immutable list containing eight elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9) | Returns an immutable list containing nine elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10) | Returns an immutable list containing ten elements. |

# 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`.

# 16.14 Java SE 9: Convenience Factory Methods for Immutable Collections(cont.)

- Map Interface's Convenience Factory Method ofEntries
  - For Maps with more than 10 key–value pairs, interface Map provides the method ofEntries.

# Common Programming Error 16.6

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