

lesson 5

Collections

Based on Chapter 19 of *Java How to Program, 7/e* (<http://www.deitel.com/books/jhttp7/>)

Learning Objectives

- Learn what collections are.
- Use class Arrays for array manipulations.
- Use the collections framework implementations.
- Use the collections framework algorithms to manipulate collections.
- Use the collections framework interfaces to program with collections polymorphically.
- Use iterators to “walk through” a collection.
- Learn about the synchronization and modifiability wrappers for collections.

Figures

```
1 // Fig. 19.2: UsingArrays.java
2 // Using Java arrays.
3 import java.util.Arrays;
4
5 public class UsingArrays
6 {
7     private int intArray[] = { 1, 2, 3, 4, 5, 6 };
8     private double doubleArray[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
9     private int filledIntArray[], intArrayCopy[];
10
11     // constructor initializes arrays
12     public UsingArrays()
13     {
14         filledIntArray = new int[ 10 ]; // create int array with 10 elements
15         intArrayCopy = new int[ intArray.length ];
16     }
17 }
```

Fig. 19.2 | Arrays class methods. (Part 1 of 3.)

```

17     Arrays.fill( filledIntArray, 7 ); // fill with 7s
18     Arrays.sort( doubleArray ); // sort doubleArray ascending
19
20     // copy array intArray into array intArrayCopy
21     System.arraycopy( intArray, 0, intArrayCopy,
22         0, intArray.length );
23 } // end UsingArrays constructor
24
25 // output values in each array
26 public void printArrays()
27 {
28     System.out.print( "doubleArray: " );
29     for ( double doubleValue : doubleArray )
30         System.out.printf( "%.1f ", doubleValue );
31
32     System.out.print( "\nintArray: " );
33     for ( int intValue : intArray )
34         System.out.printf( "%d ", intValue );
35
36     System.out.print( "\nfilledIntArray: " );
37     for ( int intValue : filledIntArray )
38         System.out.printf( "%d ", intValue );
39
40     System.out.print( "\nintArrayCopy: " );
41     for ( int intValue : intArrayCopy )
42         System.out.printf( "%d ", intValue );
43
44     System.out.println( "\n" );
45 } // end method printArrays
46
47 // find value in array intArray
48 public int searchForInt( int value )
49 {
50     return Arrays.binarySearch( intArray, value );
51 } // end method searchForInt
52
53 // compare array contents
54 public void printEquality()
55 {
56     boolean b = Arrays.equals( intArray, intArrayCopy );
57     System.out.printf( "intArray %s intArrayCopy\n",
58         ( b ? "==" : "!=" ) );
59
60     b = Arrays.equals( intArray, filledIntArray );
61     System.out.printf( "intArray %s filledIntArray\n",
62         ( b ? "==" : "!=" ) );
63 } // end method printEquality
64
65 public static void main( String args[] )
66 {
67     UsingArrays usingArrays = new UsingArrays();
68

```

Fig. 19.2 | Arrays class methods. (Part 2 of 3.)

```

69     usingArrays.printArrays();
70     usingArrays.printEquality();
71
72     int location = usingArrays.searchForInt( 5 );
73     if ( location >= 0 )
74         System.out.printf(
75             "Found 5 at element %d in intArray\n", location );
76     else
77         System.out.println( "5 not found in intArray" );
78
79     location = usingArrays.searchForInt( 8763 );
80     if ( location >= 0 )
81         System.out.printf(
82             "Found 8763 at element %d in intArray\n", location );
83     else
84         System.out.println( "8763 not found in intArray" );
85     } // end main
86 } // end class UsingArrays

```

```

doubleArray: 0.2 3.4 7.9 8.4 9.3
intArray: 1 2 3 4 5 6
filledIntArray: 7 7 7 7 7 7 7 7 7
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray

```

Fig. 19.2 | Arrays class methods. (Part 3 of 3.)

```

1 // Fig. 19.3: CollectionTest.java
2 // Using the Collection interface.
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     private static final String[] colors =
11         { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
12     private static final String[] removeColors =
13         { "RED", "WHITE", "BLUE" };
14
15     // create ArrayList, add colors to it and manipulate it
16     public CollectionTest()
17     {
18         List< String > list = new ArrayList< String >();
19         List< String > removeList = new ArrayList< String >();
20

```

Fig. 19.3 | Collection interface demonstrated via an ArrayList object. (Part 1 of 2.)

```

21 // add elements in colors array to list
22 for ( String color : colors )
23     list.add( color );
24
25 // add elements in removeColors to removeList
26 for ( String color : removeColors )
27     removeList.add( color );
28
29 System.out.println( "ArrayList: " );
30
31 // output list contents
32 for ( int count = 0; count < list.size(); count++ )
33     System.out.printf( "%s ", list.get( count ) );
34
35 // remove colors contained in removeList
36 removeColors( list, removeList );
37
38 System.out.println( "\n\nArrayList after calling removeColors: " );
39
40 // output list contents
41 for ( String color : list )
42     System.out.printf( "%s ", color );
43 } // end CollectionTest constructor
44
45 // remove colors specified in collection2 from collection1
46 private void removeColors(
47     Collection< String > collection1, Collection< String > collection2 )
48 {
49     // get iterator
50     Iterator< String > iterator = collection1.iterator();
51
52     // loop while collection has items
53     while ( iterator.hasNext() )
54
55         if ( collection2.contains( iterator.next() ) )
56             iterator.remove(); // remove current Color
57 } // end method removeColors
58
59 public static void main( String args[] )
60 {
61     new CollectionTest();
62 } // end main
63 } // end class CollectionTest

```

```

ArrayList:
MAGENTA RED WHITE BLUE CYAN
ArrayList after calling removeColors:
MAGENTA CYAN

```

Fig. 19.3 | Collection interface demonstrated via an ArrayList object. (Part 2 of 2.)

```

1 // Fig. 19.4: ListTest.java
2 // Using LinkLists.
3 import java.util.List;
4 import java.util.LinkedList;
5 import java.util.ListIterator;
6
7 public class ListTest
8 {
9     private static final String colors[] = { "black", "yellow",
10        "green", "blue", "violet", "silver" };
11     private static final String colors2[] = { "gold", "white",
12        "brown", "blue", "gray", "silver" };
13
14     // set up and manipulate LinkedList objects
15     public ListTest()
16     {
17         List< String > list1 = new LinkedList< String >();
18         List< String > list2 = new LinkedList< String >();
19
20         // add elements to list link
21         for ( String color : colors )
22             list1.add( color );
23
24         // add elements to list link2
25         for ( String color : colors2 )
26             list2.add( color );
27
28         list1.addAll( list2 ); // concatenate lists
29         list2 = null; // release resources
30         printList( list1 ); // print list1 elements
31
32         convertToUppercaseStrings( list1 ); // convert to uppercase string
33         printList( list1 ); // print list1 elements
34
35         System.out.print( "\nDeleting elements 4 to 6..." );
36         removeItems( list1, 4, 7 ); // remove items 4-7 from list
37         printList( list1 ); // print list1 elements
38         printReversedList( list1 ); // print list in reverse order
39     } // end ListTest constructor
40
41     // output List contents
42     public void printList( List< String > list )
43     {
44         System.out.println( "\nlist: " );
45
46         for ( String color : list )
47             System.out.printf( "%s ", color );
48
49         System.out.println();
50     } // end method printList
51

```

Fig. 19.4 | Lists and ListIterators. (Part I of 2.)

```

52 // locate String objects and convert to uppercase
53 private void convertToUppercaseStrings( List< String > list )
54 {
55     ListIterator< String > iterator = list.listIterator();
56
57     while ( iterator.hasNext() )
58     {
59         String color = iterator.next(); // get item
60         iterator.set( color.toUpperCase() ); // convert to uppercase
61     } // end while
62 } // end method convertToUppercaseStrings
63
64 // obtain sublist and use clear method to delete sublist items
65 private void removeItems( List< String > list, int start, int end )
66 {
67     list.subList( start, end ).clear(); // remove items
68 } // end method removeItems
69
70 // print reversed list
71 private 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
82 public static void main( String args[] )
83 {
84     new ListTest();
85 } // end main
86 } // end class ListTest

```

```

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. 19.4 | Lists and ListIterators. (Part 2 of 2.)

```

1 // Fig. 19.5: UsingToArray.java
2 // Using method toArray.

```

Fig. 19.5 | List method toArray. (Part 1 of 2.)

```

3  import java.util.LinkedList;
4  import java.util.Arrays;
5
6  public class UsingToArray
7  {
8      // constructor creates LinkedList, adds elements and converts to array
9      public UsingToArray()
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 UsingToArray constructor
29
30     public static void main( String args[] )
31     {
32         new UsingToArray();
33     } // end main
34 } // end class UsingToArray

```

```

colors:
cyan
black
blue
yellow
green
red
pink

```

Fig. 19.5 | List method toArray. (Part 2 of 2.)

Algorithm	Description
sort	Sorts the elements of a List.
binarySearch	Locates an object in a List.
reverse	Reverses the elements of a List.

Fig. 19.7 | Collections algorithms. (Part 1 of 2.)

Algorithm	Description
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 elements in the collection are equal to the specified element.
disjoint	Determines whether two collections have no elements in common.

Fig. 19.7 | Collections algorithms. (Part 2 of 2.)

```

1 // Fig. 19.8: Sort1.java
2 // Using algorithm sort.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Sort1
8 {
9     private static final String suits[] =
10         { "Hearts", "Diamonds", "Clubs", "Spades" };
11
12     // display array elements
13     public void printElements()
14     {
15         List< String > list = Arrays.asList( suits ); // create List
16
17         // output list
18         System.out.printf( "Unsorted array elements:\n%s\n", list );
19
20         Collections.sort( list ); // sort ArrayList
21
22         // output list
23         System.out.printf( "Sorted array elements:\n%s\n", list );
24     } // end method printElements
25
26     public static void main( String args[] )
27     {
28         Sort1 sort1 = new Sort1();
29         sort1.printElements();
30     } // end main
31 } // end class Sort1

```

Fig. 19.8 | Collections method sort. (Part 1 of 2.)

```

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

```

Fig. 19.8 | Collections method sort. (Part 2 of 2.)

```

1 // Fig. 19.9: Sort2.java
2 // Using a Comparator object with algorithm sort.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Sort2
8 {
9     private static final String suits[] =
10         { "Hearts", "Diamonds", "Clubs", "Spades" };
11
12     // output List elements
13     public void printElements()
14     {
15         List< String > list = Arrays.asList( suits ); // create List
16
17         // output List elements
18         System.out.printf( "Unsorted array elements:\n%s\n", list );
19
20         // sort in descending order using a comparator
21         Collections.sort( list, Collections.reverseOrder() );
22
23         // output List elements
24         System.out.printf( "Sorted list elements:\n%s\n", list );
25     } // end method printElements
26
27     public static void main( String args[] )
28     {
29         Sort2 sort2 = new Sort2();
30         sort2.printElements();
31     } // end main
32 } // end class Sort2

```

```

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

```

Fig. 19.9 | Collections method sort with a Comparator object.

```

1 // Fig. 19.10: TimeComparator.java
2 // Custom Comparator class that compares two Time2 objects.
3 import java.util.Comparator;

```

Fig. 19.10 | Custom Comparator class that compares two Time2 objects. (Part 1 of 2.)

```

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
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. 19.10 | Custom Comparator class that compares two Time2 objects. (Part 2 of 2.)

```

1 // Fig. 19.11: Sort3.java
2 // Sort a list using the custom Comparator class TimeComparator.
3 import java.util.List;
4 import java.util.ArrayList;
5 import java.util.Collections;
6
7 public class Sort3
8 {
9     public void printElements()
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
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

```

Fig. 19.11 | Collections method sort with a custom Comparator object. (Part 1 of 2.)

```

25     // output List elements
26     System.out.printf( "Sorted list elements:\n%s\n", list );
27 } // end method printElements
28
29 public static void main( String args[] )
30 {
31     Sort3 sort3 = new Sort3();
32     sort3.printElements();
33 } // end main
34 } // 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. 19.11 | Collections method sort with a custom Comparator object. (Part 2 of 2.)

```

1 // Fig. 19.14: BinarySearchTest.java
2 // Using algorithm 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     private static final String colors[] = { "red", "white",
11         "blue", "black", "yellow", "purple", "tan", "pink" };
12     private List< String > list; // ArrayList reference
13
14     // create, sort and output list
15     public BinarySearchTest()
16     {
17         list = new ArrayList< String >( Arrays.asList( colors ) );
18         Collections.sort( list ); // sort the ArrayList
19         System.out.printf( "Sorted ArrayList: %s\n", list );
20     } // end BinarySearchTest constructor
21
22     // search list for various values
23     private void search()
24     {
25         printSearchResults( colors[ 3 ] ); // first item
26         printSearchResults( colors[ 0 ] ); // middle item
27         printSearchResults( colors[ 7 ] ); // last item
28         printSearchResults( "aqua" ); // below lowest
29         printSearchResults( "gray" ); // does not exist
30         printSearchResults( "teal" ); // does not exist
31     } // end method search
32

```

Fig. 19.14 | Collections method binarySearch. (Part 1 of 2.)

```

33 // perform searches and display search result
34 private void printSearchResults( String key )
35 {
36     int result = 0;
37
38     System.out.printf( "\nSearching for: %s\n", key );
39     result = Collections.binarySearch( list, key );
40
41     if ( result >= 0 )
42         System.out.printf( "Found at index %d\n", result );
43     else
44         System.out.printf( "Not Found (%d)\n", result );
45 } // end method printSearchResults
46
47 public static void main( String args[] )
48 {
49     BinarySearchTest binarySearchTest = new BinarySearchTest();
50     binarySearchTest.search();
51 } // end main
52 } // end class BinarySearchTest

```

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. 19.14 | Collections method `binarySearch`. (Part 2 of 2.)

```

1 // Fig. 19.18: SetTest.java
2 // Using a HashSet to remove duplicates.
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

```

Fig. 19.18 | `HashSet` used to remove duplicate values from array of strings. (Part 1 of 2.)

```

9  public class SetTest
10 {
11     private static final String colors[] = { "red", "white", "blue",
12         "green", "gray", "orange", "tan", "white", "cyan",
13         "peach", "gray", "orange" };
14
15     // create and output ArrayList
16     public SetTest()
17     {
18         List< String > list = Arrays.asList( colors );
19         System.out.printf( "ArrayList: %s\n", list );
20         printNonDuplicates( list );
21     } // end SetTest constructor
22
23     // create set from array to eliminate duplicates
24     private void printNonDuplicates( Collection< String > collection )
25     {
26         // create a HashSet
27         Set< String > set = new HashSet< String >( collection );
28
29         System.out.println( "\nNonduplicates are: " );
30
31         for ( String s : set )
32             System.out.printf( "%s ", s );
33
34         System.out.println();
35     } // end method printNonDuplicates
36
37     public static void main( String args[] )
38     {
39         new SetTest();
40     } // end main
41 } // end class SetTest

```

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

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

Fig. 19.18 | HashSet used to remove duplicate values from array of strings. (Part 2 of 2.)

```

1  // Fig. 19.19: SortedSetTest.java
2  // Using TreeSet and SortedSet.
3  import java.util.Arrays;
4  import java.util.SortedSet;
5  import java.util.TreeSet;
6
7  public class SortedSetTest
8  {

```

Fig. 19.19 | Using SortedSets and TreeSets. (Part 1 of 2.)

```

9     private static final String names[] = { "yellow", "green",
10         "black", "tan", "grey", "white", "orange", "red", "green" };
11
12     // create a sorted set with TreeSet, then manipulate it
13     public SortedSetTest()
14     {
15         // create TreeSet
16         SortedSet< String > tree =
17             new TreeSet< String >( Arrays.asList( names ) );
18
19         System.out.println( "sorted set: " );
20         printSet( tree ); // output contents of tree
21
22         // get headSet based on "orange"
23         System.out.print( "\theadSet (\"orange\"): " );
24         printSet( tree.headSet( "orange" ) );
25
26         // get tailSet based upon "orange"
27         System.out.print( "\ttailSet (\"orange\"): " );
28         printSet( tree.tailSet( "orange" ) );
29
30         // get first and last elements
31         System.out.printf( "first: %s\n", tree.first() );
32         System.out.printf( "last : %s\n", tree.last() );
33     } // end SortedSetTest constructor
34
35     // output set
36     private void printSet( SortedSet< String > set )
37     {
38         for ( String s : set )
39             System.out.printf( "%s ", s );
40
41         System.out.println();
42     } // end method printSet
43
44     public static void main( String args[] )
45     {
46         new SortedSetTest();
47     } // end main
48 } // end class SortedSetTest

```

```

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. 19.19 | Using SortedSets and TreeSets. (Part 2 of 2.)

```

1 // Fig. 19.20: WordTypeCount.java
2 // Program counts the number of occurrences of each word in a string
3 import java.util.StringTokenizer;
4 import java.util.Map;
5 import java.util.HashMap;
6 import java.util.Set;
7 import java.util.TreeSet;
8 import java.util.Scanner;
9
10 public class WordTypeCount
11 {
12     private Map< String, Integer > map;
13     private Scanner scanner;
14
15     public WordTypeCount()
16     {
17         map = new HashMap< String, Integer >(); // create HashMap
18         scanner = new Scanner( System.in ); // create scanner
19         createMap(); // create map based on user input
20         displayMap(); // display map content
21     } // end WordTypeCount constructor
22
23     // create map from user input
24     private void createMap()
25     {
26         System.out.println( "Enter a string:" ); // prompt for user input
27         String input = scanner.nextLine();
28
29         // create StringTokenizer for input
30         StringTokenizer tokenizer = new StringTokenizer( input );
31
32         // processing input text
33         while ( tokenizer.hasMoreTokens() ) // while more input
34         {
35             String word = tokenizer.nextToken().toLowerCase(); // get word
36
37             // if the map contains the word
38             if ( map.containsKey( word ) ) // is word in map
39             {
40                 int count = map.get( word ); // get current count
41                 map.put( word, count + 1 ); // increment count
42             } // end if
43             else
44             {
45                 map.put( word, 1 ); // add new word with a count of 1 to map
46             } // end while
47         } // end method createMap
48
49         // display map content
50     private void displayMap()
51     {
52         Set< String > keys = map.keySet(); // get keys

```

Fig. 19.20 | HashMaps and Maps. (Part I of 2.)

```

53     // sort keys
54     TreeSet< String > sortedKeys = new TreeSet< String >( keys );
55
56     System.out.println( "Map contains:\nKey\t\tValue" );
57
58     // generate output for each key in map
59     for ( String key : sortedKeys )
60         System.out.printf( "%-10s%10s\n", key, map.get( key ) );
61
62     System.out.printf(
63         "\nsize:%d\nisEmpty:%b\n", map.size(), map.isEmpty() );
64 } // end method displayMap
65
66 public static void main( String args[] )
67 {
68     new WordTypeCount();
69 } // end main
70 } // end class WordTypeCount

```

```

Enter a string:
To be or not to be: that is the question Whether 'tis nobler to suffer
Map contains:
Key                Value
'tis                1
be                  1
be:                 1
is                  1
nobler              1
not                 1
or                  1
question            1
suffer              1
that                1
the                 1
to                  3
whether               1

size:13
isEmpty:false

```

Fig. 19.20 | HashMaps and Maps. (Part 2 of 2.)

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 )

```

Fig. 19.21 | Synchronization wrapper methods. (Part 1 of 2.)

```
public static method headers
```

```
< K, V > Map< K, V > synchronizedMap( Map< K, V > m )
```

```
< K, V > SortedMap< K, V > synchronizedSortedMap( SortedMap< K, V > m )
```

Fig. 19.21 | Synchronization wrapper methods. (Part 2 of 2.)

```
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. 19.22 | Unmodifiable wrapper methods.