# Generic Collections – Part I

## Introduction

- Java collections framework
  - prebuilt data structures
  - interfaces and methods for manipulating those data structures

# 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.
- Figure 20.1 lists some of the interfaces of the collections framework.
- 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` | 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. |

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

## 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`.
- 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`.



## Type-Wrapper Classes for Primitive Types (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.

# 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 ];
```



# Interface `Collection` and Class `Collections`

- Interface `Collection` is the root interface from which interfaces `Set`, `Queue` and `List` are derived.
- Interface `Set` defines a collection that does not contain duplicates.
- Interface `Queue` defines a collection that represents a waiting line.
- 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.*

## Lists

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



## 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.
- We discuss the architecture of linked lists in Chapter 22.
- 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.
- For this reason, `ArrayList` is typically preferred over `Vector` in programs that do not share a collection among threads.

**Performance Tip 20.1**

*`ArrayList`s behave like `Vector`s without synchronization and therefore execute faster than `Vector`s because `ArrayList`s do not have the overhead of thread synchronization.*



**Software Engineering Observation 20.3**

*`LinkedList`s 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.*

# 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.
  - 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
```

Good practice to reference a collection via an interface-type variable—easier to change the collection later

**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
```

Method works with any `Collection`

```
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* `ConcurrentModificationException`*s. For this reason, iterators are said to be "fail fast."*



## 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`.
- `List-Iterator` 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` obtaina a portion of a `List`.
  - This is a so-called range-view method, which enables the program to view a portion of the list.

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

**Fig. 20.3** | Lists, LinkedLists and ListIterators. (Part 1 of 4.)

```
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
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
```

**Fig. 20.3** | Lists, LinkedLists and ListIterators. (Part 2 of 4.)



```
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
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
```

subList returns a view into a List

**Fig. 20.3** | Lists, LinkedLists and ListIterators. (Part 3 of 4.)

```
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
```

```
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 4 of 4.)



# `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. 20.4: 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
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: " );
```

Preallocating the array allows `toArray` to simply copy elements into the array

**Fig. 20.4** | Viewing arrays as `Lists` and converting `Lists` to arrays. (Part 1 of 2.)



```
25
26       for ( String color : colors )
27          System.out.println( color );
28    } // end main
29 } // end class UsingToArray
```

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

**Fig. 20.4** | Viewing arrays as `Lists` and converting `Lists` to arrays. (Part 2 of 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.*

# Collections Methods

- Class `Collections` provides several high-performance algorithms for manipulating collection elements.
- The algorithms are implemented as `static` methods.



| 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.*



## 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
```

`list` elements must be `Comparable`

**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.)

# 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
```

**Fig. 20.10** | Card shuffling and dealing with `Collections` method `shuffle`. (Part 1 of 5.)

```
23
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
```

Shuffles the contents of a collection

**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.)

# 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 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.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
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 );
```

**Fig. 20.11** | `Collections` methods `reverse`, `fill`, `copy`, `max` and `min`. (Part 2 of 3.)



```
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
```

```
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 3 of 3.)

# 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.
  - Method `binarySearch` determines this negative value by first calculating the insertion point and making its sign negative.
  - Then, `binarySearch` subtracts 1 from the insertion point to obtain the return value, which guarantees that method `binarySearch` returns positive numbers (>= 0) if and only if the object is found.



```
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
21       // search list for various values
22       printSearchResults( list, colors[ 3 ] ); // first item
23       printSearchResults( list, colors[ 0 ] ); // middle item
```

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

```
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
```

Collection must be sorted first

**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.)

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

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