Java Programming

Generic Collections

Generic Collection

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

- 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.

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.

Fig. 20.1 | Some collections-framework interfaces.

Class `ArrayList`

Method	Description
`add`	Adds an element to the end of the `ArrayList`.
`clear`	Removes all the elements from the `ArrayList`.
`contains`	Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`.
`get`	Returns the element at the specified index.
`indexOf`	Returns the index of the first occurrence of the specified element in the `ArrayList`.
`remove`	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
`size`	Returns the number of elements stored in the `ArrayList`.
`trimToSize`	Trims the capacity of the `ArrayList` to current number of elements.

Fig. 7.23 | Some methods and properties of class `ArrayList<T>`.

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

- 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.
- 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.

ArrayList and Iterator

- `List` method `add` adds an item to the end of a list.
- `List` method `size` returns the number of elements.
- `List` method `get` retrieves an individual element's value from the specified index.
- `Collection` method `iterator` gets an `Iterator` for a `Collection`.
- `Iterator-` method `hasNext` determines whether 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`.

ArrayList and Iterator

```
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.)

ArrayList and Iterator

```
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.)

ArrayList and Iterator

```
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.)

LinkedList

- `List` method `addAll` appends all elements of a collection to the end of a `List`.
- `List` method `listIterator` gets A `List`'s bidirectional iterator.
- `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`.

LinkedList

- `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.
- `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.

LinkedList

- 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.

LinkedList

```
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.)

LinkedList

```
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
```

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

LinkedList

```
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.)

LinkedList

```
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.)

LinkedList

```
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.)

LinkedList

```
 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
```

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

`LinkedList`

- `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`.

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.

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.
  - 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.

Method sort

```
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.)

Method sort

```
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.)

Method sort

```
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.)

Method sort

```
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.)

Method sort

```
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.)

Method sort

```
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
```

```
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.)

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.)

Method shuffle

```
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.)

Method shuffle

```
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.)

Method shuffle

```
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.)

Method shuffle

```
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`.

Methods reverse, fill, copy, max and min

```
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.)

Methods reverse, fill, copy, max and min

```
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.)

Methods reverse, fill, copy, max and min

```
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.)

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.

Method binarySearch

```
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.)

Method binarySearch

```
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.)

Method binarySearch

```
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.

Stack Class of Package java.util

- Class Stack in the Java utilities package (java-.util) extends class Vector to implement a stack data structure.
- Stack method push adds a Number object to the top of the stack.
- Any integer literal that has the suffix L is a long value.
- An integer literal without a suffix is an int value.
- Any floating-point literal that has the suffix F is a float value.
- A floating-point literal without a suffix is a double value.
- Stack method pop removes the top element of the stack.
  - If there are no elements in the Stack, method pop throws an EmptyStackException, which terminates the loop.
- Method peek returns the top element of the stack without popping the element off the stack.
- Method isEmpty determines whether the stack is empty.

Stack Class of Package java.util

```
1  // Fig. 20.14: StackTest.java
2  // Stack class of package java.util.
3  import java.util.Stack;
4  import java.util.EmptyStackException;
5
6  public class StackTest
7  {
8     public static void main( String[] args )
9     {
10       Stack< Number > stack = new Stack< Number >(); // create a Stack
11
12       // use push method
13       stack.push( 12L ); // push long value 12L
14       System.out.println( "Pushed 12L" );
15       printStack( stack );
16       stack.push( 34567 ); // push int value 34567
17       System.out.println( "Pushed 34567" );
18       printStack( stack );
19       stack.push( 1.0F ); // push float value 1.0F
20       System.out.println( "Pushed 1.0F" );
21       printStack( stack );
22       stack.push( 1234.5678 ); // push double value 1234.5678
23       System.out.println( "Pushed 1234.5678 " );
24       printStack( stack );
```

Fig. 20.14 | Stack class of package `java.util`. (Part 1 of 4.)

Stack Class of Package java.util

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

Fig. 20.14 | Stack class of package java.util. (Part 2 of 4.)

Stack Class of Package java.util

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

Fig. 20.14 | Stack class of package java.util. (Part 3 of 4.)

Stack Class of Package java.util

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

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

Fig. 20.14 | Stack class of package java.util. (Part 4 of 4.)

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.