16.6 Lists

A `List` (sometimes called a **sequence**) is a `Collection` of elements in sequence that can contain duplicate elements. Like array indices, `List` indices are zero based (i.e., the first element's index is zero). 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` and `LinkedList`. Autoboxing occurs when you add primitive-type values to objects of these classes, because they store only references to objects. Class `ArrayList` is a resizable-array implementations of `List`. Inserting an element between existing elements of an `ArrayList` is an *inefficient* operation—all elements after the new one must be moved out of the way, which could be an expensive operation in a collection with a large number of elements. A `LinkedList` enables *efficient* insertion (or removal) of elements in the middle of a collection, but is much less efficient than an `ArrayList` for jumping to a specific element in the collection. We discuss the architecture of linked lists in Chapter 21.

The following two subsections demonstrate the `List` and

`Collection` capabilities. Section 16.6.1 removes elements from an `ArrayList` with an `Iterator`. Section 16.6.2 uses `ListIterator` and several `List`- and `LinkedList`-specific methods.

16.6.1 `ArrayList` and `Iterator`

Figure 16.2 uses an `ArrayList` (introduced in Section 7.16) to demonstrate several capabilities of interface `Collection`. The program places two `Color` arrays in `ArrayList`s and uses an `Iterator` to remove elements in the second `ArrayList` collection from the first.

```
      1   // Fig. 16.2: CollectionTest.java
 2   // Collection interface demonstrated via an Arra
            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      public static void main(String[] args) {
10         // add elements in colors array to list
11         String[] colors = {"MAGENTA", "RED", "WHIT
12         List<String> list = new ArrayList<String>(
                        13
     14         for (String color : colors) {
15            list.add(color); // adds color to end o
                    16         }
                         17
18         // add elements in removeColors array to r
```

```
19          String[] removeColors = {"RED", "WHITE", "
20          List<String> removeList = new ArrayList<St
                      21
 22           for (String color : removeColors) {
       23              removeList.add(color);
                    24          }
                      25
        26          // output list contents
  27           System.out.println("ArrayList: ");
                      28
29          for (int count = 0; count < list.size(); c
30             System.out.printf("%s ", list.get(count
                    31          }
                      32
33          // remove from list the colors contained i
    34           removeColors(list, removeList);
                      35
        36          // output list contents
37          System.out.printf("%n%nArrayList after cal
                      38
      39           for (String color : list) {
  40              System.out.printf("%s ", color);
                    41          }
                      42       }
                      43
44       // remove colors specified in collection2 fro
45       private static void removeColors(Collection<S
   46           Collection<String> collection2) {
            47          // get iterator
48          Iterator<String> iterator = collection1.it
                      49
  50          // loop while collection has items
     51           while (iterator.hasNext()) {
52              if (collection2.contains(iterator.next(
53                 iterator.remove(); // remove current
                  54             }
                    55          }
                      56       }
                        57   }
```

```
                    ArrayList:
              MAGENTA RED WHITE BLUE CYAN

        ArrayList after calling removeColors:
                  MAGENTA CYAN
```

Fig. 16.2

`Collection` interface demonstrated via an `ArrayList` object.

Lines 11 and 19 declare and initialize `String` arrays `colors` and `removeColors`. Lines 12 and 20 create `ArrayList<String>` objects and assign their references to `List<String>` variables `list` and `removeList`, respectively. Recall that `ArrayList` is a *generic* class, so we can specify a *type argument* (`String` in this case) to indicate the type of the elements in each list. Because you specify the type to store in a collection at compile time, generic collections provide compile-time *type safety* that allows the compiler to catch attempts to use invalid types. For example, you cannot store `Employee`s in a collection of `String`s.

Lines 14–16 populate `list` with `String`s stored in array `colors`, and lines 22–24 populate `removeList` with `String`s stored in array `removeColors` using `List` **method** `add`, which adds elements to the end of the `List`. Lines 29–31 output each element of `list`. Line 29 calls

`List` **method** `size` to get the number of elements in the `ArrayList`. Line 30 uses `List` **method** `get` to retrieve individual element values. Lines 29–31 also could have used the enhanced `for` statement.

Line 34 calls method `removeColors` (lines 45–56), passing `list` and `removeList` as arguments. Method `removeColors` deletes the `String`s in `removeList` from the `String`s in `list`. Lines 39–41 print `list`'s elements after `removeColors` completes its task.

Method `removeColors` declares two `Collection<String>` parameters (lines 45–46)—any two `Collection`s containing `String`s can be passed as arguments. The method accesses the elements of the first `Collection` (`collection1`) via an `Iterator`. Line 48 calls `Collection` **method** `iterator` to get an `Iterator` for the `Collection`. Interfaces `Collection` and `Iterator` are generic types. The loop-continuation condition (line 51) calls `Iterator` **method** `hasNext` to determine whether there are more elements to iterate through. Method `hasNext` returns `true` if another element exists and `false` otherwise.

The `if` condition in line 52 calls `Iterator` **method** `next` to obtain a reference to the next element, then uses method `contains` of the second `Collection` (`collection2`) to determine whether `collection2` contains the element returned by `next`. If so, line 53 calls `Iterator` **method** `remove` to remove the element from the `Collection`

`collection1`.

Common Programming Error 16.1

If a collection is modified by one of its methods after an iterator is created for that collection, the iterator immediately becomes invalid—any operation performed with the iterator fails immediately and throws a `ConcurrentModificationException`. *For this reason, iterators are said to be "fail fast." Fail-fast iterators help ensure that a modifiable collection is not manipulated by two or more threads at the same time, which could corrupt the collection. In Chapter 23, Concurrency, you'll learn about concurrent collections (package* `java.util.concurrent`*) that can be safely manipulated by multiple concurrent threads.*

Software Engineering Observation 16.3

We refer to the `ArrayList`*s in this example via* `List` *variables. This makes our code more flexible and easier to modify—if we later determine that* `LinkedList`*s would be more appropriate, only the lines where we created the*

`ArrayList` *objects (lines 12 and 20) need to be modified. In general, when you create a collection object, refer to that object with a variable of the corresponding collection interface type. Similarly, implementing method* `removeColors` *to receive* `Collection` *references enables the method to be used with any collection that implements the interface* `Collection`.

Type Inference with the <> Notation

Lines 12 and 20 specify the type stored in the `ArrayList` (that is, `String`) on the left and right sides of the initialization statements. You also can use *type inferencing* with `<>`—known as the **diamond notation**—in statements that declare and create generic type variables and objects. For example, line 12 can be written as:

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

In this case, 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. We'll use this syntax for the remaining examples in this chapter.

16.6.2 `LinkedList`

Figure 16.3 demonstrates various operations on `LinkedList`s. The program creates two `LinkedList`s of `String`s. The elements of one `List` are added to the other. Then all the `String`s are converted to uppercase, and a range of elements is deleted.

```
1  // Fig. 16.3: ListTest.java
2  // Lists, LinkedLists and ListIterators.
3  import java.util.List;
4  import java.util.LinkedList;
5  import java.util.ListIterator;
6
7  public class ListTest {
8     public static void main(String[] args) {
9        // add colors elements to list1
10       String[] colors =
11          {"black", "yellow", "green", "blue", "v
12       List<String> list1 = new LinkedList<>();
13
14       for (String color : colors) {
15          list1.add(color);
16       }
17
18       // add colors2 elements to list2
19       String[] colors2 =
20          {"gold", "white", "brown", "blue", "gra
21       List<String> list2 = new LinkedList<>();
22
23       for (String color : colors2) {
24          list2.add(color);
25       }
26
27       list1.addAll(list2); // concatenate lists
28       list2 = null; // release resources
```

```
29          printList(list1); // print list1 elements
                      30
31          convertToUppercaseStrings(list1); // conve
32          printList(list1); // print list1 elements
                      33
34          System.out.printf("%nDeleting elements 4 t
35          removeItems(list1, 4, 7); // remove items
36          printList(list1); // print list1 elements
37          printReversedList(list1); // print list in
                    38      }
                      39
           40      // output List contents
41       private static void printList(List<String> li
    42          System.out.printf("%nlist:%n");
                      43
       44          for (String color : list) {
  45              System.out.printf("%s ", color);
                    46          }
                      47
          48          System.out.println();
                     49      }
                      50
51       // locate String objects and convert to upper
52       private static void convertToUppercaseStrings
53          ListIterator<String> iterator = list.listI
                      54
      55          while (iterator.hasNext()) {
56              String color = iterator.next(); // get
57              iterator.set(color.toUpperCase()); // c
                    58          }
                     59      }
                      60
61       // obtain sublist and use clear method to del
62       private static void removeItems(List<String>
          63          int start, int end) {
64          list.subList(start, end).clear(); // remov
                    65      }
                      66
           67      // print reversed list
68       private static void printReversedList(List<St
```

```
69          ListIterator<String> iterator = list.listI
                        70
71          System.out.printf("%nReversed List:%n");
                        72
     73           // print list in reverse order
    74          while (iterator.hasPrevious()) {
75              System.out.printf("%s ", iterator.previ
                    76          }
                      77      }
                        78   }
```

```
                        list:
black yellow green blue violet silver gold white brow

                        list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROW

             Deleting elements 4 to 6…
                        list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

                    Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```

Fig. 16.3

`List`s, `LinkedList`s and `ListIterator`s.

Lines 12 and 21 create `LinkedList`s `list1` and `list2` of type `String`. `LinkedList` is a generic class that has one type parameter, for which we specify the type argument `String` in this example. Lines 14–16 and 23–25 call `List`

method `add` to *append* elements from arrays `colors` and `colors2` to the *ends* of `list1` and `list2`, respectively.

Line 27 calls `List` **method** `addAll` to *append all elements* of `list2` to the end of `list1`. Line 28 sets `list2` to `null`, because `list2` is no longer needed. Line 29 calls method `printList` (lines 41–49) to output `list1`'s contents. Line 31 calls method `convertToUppercaseStrings` (lines 52–59) to convert each `String` element to uppercase, then line 32 calls `printList` again to display the modified `String`s. Line 35 calls method `removeItems` (lines 62–65) to remove the range of elements starting at index `4` up to, but not including, index `7` of the list. Line 37 calls method `printReversedList` (lines 68–77) to print the list in reverse order.

Method `convertToUppercaseStrings`

Method `convertToUppercaseStrings` (lines 52–59) changes lowercase `String` elements in its `List` argument to uppercase `String`s. Line 53 calls `List` **method** `listIterator` to get the `List`'s **bidirectional iterator** (i.e., one that can traverse a `List` *backward* or *forward*). `ListIterator` is also a generic class. In this example, the `ListIterator` references `String` objects, because

method `listIterator` is called on a `List` of `String`s. Line 55 calls method `hasNext` to determine whether the `List` *contains another element*. Line 56 gets the next `String` in the `List`. Line 57 calls `String` **method** `toUpperCase` to get an uppercase version of the `String` and calls `ListIterator` **method** `set` to replace the current `String` to which `iterator` refers with the `String` returned by method `toUpperCase`. Like method `toUpperCase`, `String` **method** `toLowerCase` returns a lowercase version of the `String`.

Method `removeItems`

Method `removeItems` (lines 62–65) *removes a range of items* from the list. Line 64 calls `List` **method** `subList` to obtain a portion of the `List` (called a **sublist**). This is called a **range-view method**, which enables the program to view a portion of the list. The sublist is simply a view into the `List` on which `subList` is called. Method `subList` takes as arguments the beginning and ending index for the sublist. The ending index is *not* part of the range of the sublist. In this example, line 35 passes `4` for the beginning index and `7` for the ending index to `subList`. The sublist returned is the set of elements with indices `4` through `6`. Next, the program calls `List` **method** `clear` on the sublist to remove the elements of the sublist from the `List`. Any changes made to a sublist are also made to the original `List`.

Method `printReversedList`

Method `printReversedList` (lines 68–77) prints the list backward. Line 69 calls `List` method `listIterator` with the starting position as an argument (in our case, the last element in the list) to get a *bidirectional iterator* for the list. `List` **method** `size` returns the number of items in the `List`. The `while` condition (line 74) calls `ListIterator`**'s** `hasPrevious` **method** to determine whether there are more elements while traversing the list *backward*. Line 75 calls `ListIterator`**'s** `previous` **method** to get the previous element from the list and outputs it to the standard output stream.

Views into Collections and `Arrays` Method `asList`

Class `Arrays` provides `static` method `asList` to *view* an array (sometimes called the **backing 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. The next example demonstrates how to create a `LinkedList` with a `List` view of an array, because we cannot pass the array to a `LinkedList` constructor. Sorting array elements with a `List` view is demonstrated in Fig. 16.7. 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 (such as adding or removing elements) result in an `UnsupportedOperationException`.

Viewing Arrays as `Lists` and Converting `Lists` to Arrays

Figure 16.4 uses `Arrays` method `asList` to view an array as a `List` and uses `List` **method** `toArray` to get an array from a `LinkedList` collection. The program calls method `asList` to create a `List` view of an array, which is used to initialize a `LinkedList` object, then adds a series of `String`s to the `LinkedList` and calls method `toArray` to obtain an array containing references to the `String`s.

```
   1   // Fig. 16.4: UsingToArray.java
2   // Viewing arrays as Lists and converting Lists
   3   import java.util.LinkedList;
     4   import java.util.Arrays;
                  5
    6   public class UsingToArray {
 7     public static void main(String[] args) {
8       String[] colors = {"black", "blue", "yello
9       LinkedList<String> links = new LinkedList<
```

```
                    10
11          links.addLast("red"); // add as last item
 12          links.add("pink"); // add to the end
13          links.add(3, "green"); // add at 3rd index
14          links.addFirst("cyan"); // add as first it
                    15
16          // get LinkedList elements as an array
17          colors = links.toArray(new String[links.si
                    18
    19          System.out.println("colors: ");
                    20
     21          for (String color : colors) {
     22             System.out.println(color);
                    23          }
                      24       }
                        25    }
```

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

Fig. 16.4

Viewing arrays as `List`s and converting `List`s to arrays.

Line 9 constructs a `LinkedList` of `String`s containing the elements of array `colors`. `Arrays` method `asList` returns

a `List` view of the array, then uses that to initialize the `LinkedList` with its constructor that receives a `Collection` as an argument (a `List` *is a* `Collection`). Line 11 calls `LinkedList` **method** `addLast` to add `"red"` to the end of `links`. Lines 12–13 call `LinkedList` **method** `add` to add `"pink"` as the last element and `"green"` as the element at index `3` (i.e., the fourth element). Method `addLast` works identically to the single-argument `add` method. Line 14 calls `LinkedList` **method** `addFirst` to add `"cyan"` as the new first item in the `LinkedList`. The `add` operations are permitted because they operate on the `LinkedList` object, not the view returned by `asList`. [*Note:* When `"cyan"` is added as the first element, `"green"` becomes the fifth element in the `LinkedList`.]

Line 17 calls interface `List`'s `toArray` method to get a `String` array from `links`. The array is a copy of the list's elements—modifying the array's contents does *not* modify the list. The array passed to method `toArray` is of the type that you'd like method `toArray` to return. If the number of elements in that array is greater than or equal to the number of elements in the `LinkedList`, `toArray` copies the list's elements into its array argument and returns that array. If the `LinkedList` has more elements than the number of elements in the array passed to `toArray`, `toArray` *allocates a new array* of the same type it receives as an argument, *copies* the list's elements into the new array and returns the new array.

Common Programming Error 16.2

Passing an array that contains data as `toArray`*'s argument can cause logic errors. If the array's number of elements 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 array's number of elements is greater than the number of elements in the list, the array's elements (starting at index zero) are overwritten with the list's elements. The first element of the remainder of the array is set to null to indicate the end of the list.*