7.16 Introduction to Collections and Class `ArrayList`

The Java API provides several predefined data structures, called **collections**, used to store groups of related objects in memory. These classes provide efficient methods that organize, store and retrieve your data *without* requiring knowledge of how the data is being *stored*. This reduces application-development time.

You've used arrays to store sequences of objects. Arrays do not automatically change their size at execution time to accommodate additional elements. The collection class `ArrayList<E>` (package `java.util`) provides a convenient solution to this problem—it can *dynamically* change its size to accommodate more elements. The `E` (by convention) is a *placeholder*—when declaring a new `ArrayList`, replace it with the type of elements that you want the `ArrayList` to hold. For example,

```
ArrayList<String> list;
```

declares `list` as an `ArrayList` collection that can store only `String`s. Classes with this kind of placeholder that can

be used with any type are called **generic classes**. *Only reference types can be used to declare variables and create objects of generic classes*. However, Java provides a mechanism—known as *boxing*—that allows primitive values to be wrapped as objects for use with generic classes. So, for example,

```
ArrayList<Integer> integers;
```

declares `integers` as an `ArrayList` that can store only `Integer`s. When you place an `int` value into an `ArrayList<Integer>`, the `int` value is *boxed* (wrapped) as an `Integer` object, and when you get an `Integer` object from an `ArrayList<Integer>`, then assign the object to an `int` variable, the `int` value inside the object is *unboxed* (unwrapped).

Additional generic collection classes and generics are discussed in Chapters 16 and 20, respectively. Figure 7.23 shows some common methods of class `ArrayList<E>`.

Fig. 7.23

Some methods of class `ArrayList<E>`.

Method	Description
`add`	Overloaded to add an element to the *end* of the `ArrayList` or at a specific index in 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 the current number of elements.

Demonstrating an `ArrayList<String>`

Figure 7.24 demonstrates some common `ArrayList` capabilities. Line 8 creates a new empty `ArrayList` of `String`s with a default initial capacity of 10 elements. The capacity indicates how many items the `ArrayList` can hold *without growing*. `ArrayList` is implemented using a conventional array behind the scenes. When the `ArrayList` grows, it must create a larger internal array and *copy* each element to the new array. This is a time-consuming operation. It would be inefficient for the `ArrayList` to grow each time an element is added. Instead, it grows only when an element is added *and* the number of elements is equal to the capacity—i.e., there's no space for the new element.

```
1  // Fig. 7.24: ArrayListCollection.java
2  // Generic ArrayList<E> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection {
6     public static void main(String[] args) {
7        // create a new ArrayList of Strings with a
8        ArrayList<String> items = new ArrayList<Str
9
10       items.add("red"); // append an item to the
11       items.add(0, "yellow"); // insert "yellow"
12
13          // header
14       System.out.print(
15          "Display list contents with counter-con
16
17       // display the colors in the list
18       for (int i = 0; i < items.size(); i++) {
19          System.out.printf(" %s", items.get(i));
20       }
21
22       // display colors using enhanced for in th
23       display(items,
24          "%nDisplay list contents with enhanced
25
26       items.add("green"); // add "green" to the
27       items.add("yellow"); // add "yellow" to th
28       display(items, "List with two new elements
29
30       items.remove("yellow"); // remove the firs
31       display(items, "Remove first instance of y
32
33       items.remove(1); // remove item at index 1
34       display(items, "Remove second list element
35
36       // check if a value is in the List
37       System.out.printf("\"red\" is %sin the lis
38          items.contains("red") ? "": "not ");
39
40       // display number of elements in the List
```

```
41          System.out.printf("Size: %s%n", items.size
42       }
43
44       // display the ArrayList's elements on the co
45       public static void display(ArrayList<String>
46          System.out.printf(header); // display head
47
48          // display each element in items
49             for (String item : items) {
50                System.out.printf(" %s", item);
51             }
52
53             System.out.println();
54       }
55    }
```

```
Display list contents with counter-controlled loop: y
Display list contents with enhanced for statement: ye
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
  Remove second list element (green): red yellow
              "red" is in the list
                    Size: 2
```

Fig. 7.24

Generic `ArrayList<E>` collection demonstration.

The `add` method adds elements to the `ArrayList` (lines 10–11). The `add` method with *one* argument *appends* its argument to the *end* of the `ArrayList`. The `add` method with *two*

arguments *inserts* a new element at the specified *position*. The first argument is an index. As with arrays, collection indices start at zero. The second argument is the *value* to insert at that *index*. The indices of all subsequent elements are incremented by one. Inserting an element is usually slower than adding an element to the end of the `ArrayList`.

Lines 18–20 display the items in the `ArrayList`. Method `size` returns the number of elements currently in the `ArrayList`. Method `get` (line 19) obtains the element at a specified index. Lines 23–24 display the elements again by invoking method `display` (defined at lines 45–54). Lines 26–27 add two more elements to the `ArrayList`, then line 28 displays the elements again to confirm that the two elements were added to the *end* of the collection.

The `remove` method is used to remove an element with a specific value (line 30). It removes only the first such element. If no such element is in the `ArrayList`, `remove` does nothing. An overloaded version of the method removes the element at the specified index (line 33). When an element is removed, the indices of any elements after the removed element decrease by one.

Line 38 uses the `contains` method to check if an item is in the `ArrayList`. The `contains` method returns `true` if the element is found in the `ArrayList`, and `false` otherwise. The method compares its argument to each element of the `ArrayList` in order, so using `contains` on a large `ArrayList` can be *inefficient*. Line 41 displays the

`ArrayList`'s size.

Diamond (<>) Notation for Creating an Object of a Generic Class

Consider line 8 of Fig. 7.24:

```
ArrayList<String> items = new ArrayList<String>();
```

Notice that `ArrayList<String>` appears in the variable declaration *and* in the class instance creation expression. The **diamond (<>) notation** simplifies statements like this. Using <> in a class instance creation expression for an object of a *generic* class tells the compiler to determine what belongs in the angle brackets. The preceding statement can be written as:

```
ArrayList<String> items = new ArrayList<>();
```

When the compiler encounters the diamond (<>) in the class instance creation expression, it uses the declaration of variable `items` to determine the `ArrayList`'s element type (`String`)—this is known as *inferring the element type*.