The ArrayList Class

collection

A *collection* is a group of related objects, or elements, that are stored together as a single unit. An array is an example of a collection. Java also contains a *collections framework*, which provides classes for implementing collections. One such class is the ArrayList class, which includes methods for adding and deleting elements and finding an element.

collections framework

TIP The ArrayList class is a data structure. Data structures are discussed further in Chapter 14.

Class ArrayList (java.util.ArrayList)

Method	
`add(int index, element)`	inserts `element` at `index` position of the array. Existing elements are shifted to the next position up in the array.
`add(element)`	adds `element` to the end of the array.
`get(int index)`	returns the `element` at `index` position in the array.
`indexOf(obj)`	returns the `index` of the first element matching `obj` using the equals() method of the object's class to determine equality between the element and the object.
`remove(int index)`	removes the element at `index` position in the array.
`set(int index, element)`	replaces the element at `index` position with `element`.
`size()`	returns the number of elements in the array.

Data Structure Analysis

Data structure analysis includes measuring the efficiency of a data structure's operations. For example, adding an object to the end of an ArrayList data structure requires a single operation, which can be written as O(1). However, adding an object to the beginning of the array requires all the existing objects first be moved up one position. For an array with *n* objects, adding an object to the front of the array requires *n* operations, which can be written as O(n). This is a much less efficient operation.

dynamic array

The ArrayList class implements a dynamic array. A *dynamic array* varies in size during run time and is used in applications where the size of an array may need to grow or shrink. An ArrayList object shifts elements up one position when a new element is added at a specific index. Elements added to the end of the ArrayList do not move existing elements. Removing an element from an ArrayList also shifts elements as necessary to close the gap left by the removed element.

When using an ArrayList object, it is important to understand that only objects, not primitive types, can be stored. Because the indexOf() method compares its object parameter to each element of the array, it is important that the object's class has an appropriately overridden equals() method.

equals()

generics

Collections, such as ArrayList, make use of *generics* for communicating to the compiler the type of data stored. For example, to declare an ArrayList of String objects, a statement using generics appears similar to:

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

type parameter

The `<String>` part of the statement is a *type parameter* informing the compiler of what type of objects the ArrayList can contain. The type parameter is read "of *Type*" or in this case "of String."

The following application creates an ArrayList object, adds elements, removes an element, and then displays the remaining elements:

```
import java.util.ArrayList;

public class TestArrayList {

  public static void main(String[] args) {
    ArrayList<String> myStrings = new ArrayList<String>();

    myStrings.add("Kermit");
    myStrings.add("Lucille");
    myStrings.add("Sammy");
    myStrings.add("Roxy");
    myStrings.add("Myah");

    myStrings.remove(3);

    for (String name : myStrings) {
      System.out.println(name);
    }
  }
}
```

TIP The `for-each` statement is not a safe structure for finding and removing elements from an ArrayList.

The ArrayList declaration does not require an array size. An ArrayList object grows and shrinks automatically as elements are added and removed. The `for-each` statement traverses the ArrayList.

The TestArrayList application displays the output:

```
Kermit
Lucille
Sammy
Myah
```

Wrapper Classes

Primitive data types cannot be directly stored in an ArrayList because the elements in an ArrayList must be objects. The Integer and Double classes, provided with Java, are used to "wrap" primitive values in an object. The Integer and Double *wrapper classes* include methods for comparing objects and for returning, or "unwrapping", the primitive value stored by the object:

TIP Java also includes the Character, Boolean, Byte, Short, Long, and Float wrapper classes.

Class Integer (java.lang.Integer)

Method	
`compareTo(Integer intObject)`	returns 0 when the Integer object value is the same as `intObject`. A negative `int` is returned when the Integer object is less than `intObject`, and a positive `int` is returned when the Integer object is greater than `intObject`.
`intValue()`	returns the `int` value of the Integer object.

Class Double (java.lang.Double)

Method	

TIP The Integer and Double classes implement the Comparable interface, introduced in Chapter 9.

`compareTo(Double doubleObject)`	returns 0 when the Double object value is the same as `doubleObject`. A negative `int` is returned when the Double object is less than `doubleObject`, and a positive `int` is returned when the Double object is greater than `doubleObject`.
`doubleValue()`	returns the `double` value of the Double object.

The Integer and Double wrapper classes are in the java.lang package. Therefore applications do not require an `import` statement to use the wrapper classes.

The DataArrayList application creates an ArrayList of Integer values, compares the values, and then sums the elements:

```
import java.util.ArrayList;

public class DataArrayList {

 public static void main(String[] args) {
   ArrayList<Integer> numbers = new ArrayList<Integer>();
   Integer element, element1, element2;
   int sum = 0;

   numbers.add(new Integer(5));
   numbers.add(new Integer(2));

   /* compare values */
   element1 = numbers.get(0);
   element2 = numbers.get(1);
   if (element1.compareTo(element2) == 0) {
     System.out.println("The elements have the same value.");
   } else if (element1.compareTo(element2) < 0) {
     System.out.println("element1 value is less than element2.");
   } else {
     System.out.println("element1 value is greater than element2.");
   }

   /* sum values */
   for (Integer num : numbers) {
     element = num;
     sum += element.intValue();      //use int value for sum
   }
   System.out.println("Sum of the elements is: " + sum);
 }
}
```

In the first `numbers.add()` statement above, the `new` operator allocates memory and returns a reference to the Integer object that stores the value 5. A second statement performs the same type of action to add another Integer object to the ArrayList.

The values stored in the elements are compared using the compareTo() method. Note that the elements are first assigned to Integer objects `element1` and `element2`.

Finally, the `for-each` statement traverses the ArrayList elements and sums their values. Each element must be "unwrapped" so that the `int` value of each Integer object is used to update `sum`.

Autoboxing and Auto-Unboxing

Java 5 includes autoboxing and auto-unboxing features to eliminate the additional code needed to wrap and unwrap primitives and their corresponding object types. For example, with this automated feature, the statements:

```
nums.add(2);
sum += nums.get(0);
```

can be used instead of:

```
Integer num;
nums.add(new Integer(2));
num = nums.get(0);
sum += num.intValue();
```

The DataArrayList displays the output:

```
element1 value is greater than element2.
Sum of the elements is: 7
```