# The Array List Class

Slides prepared by Rose Williams, *Binghamton University*

Kenrick Mock, *University of Alaska Anchorage*

# The **`ArrayList`** Class

- **`ArrayList`** is a class in the standard Java libraries
  - Unlike arrays, which have a fixed length once they have been created, an **`ArrayList`** is an object that can grow and shrink while your program is running
- In general, an **`ArrayList`** serves the same purpose as an array, except that an **`ArrayList`** can change length while the program is running

# The **`ArrayList`** Class

- The class **`ArrayList`** is implemented using an array as a private instance variable
  - When this hidden array is full, a new larger hidden array is created, and the data is transferred to this new array

# The **`ArrayList`** Class

- Why not always use an **`ArrayList`** instead of an array?

1. An **`ArrayList`** is less efficient than an array
2. It does not have the convenient square bracket notation
3. The base type of an **`ArrayList`** must be a class type (or other reference type): it cannot be a primitive type

– This last point is less of a problem now that Java provides automatic boxing and unboxing of primitives

# Using the **`ArrayList`** Class

- In order to make use of the **`ArrayList`** class, it must first be imported from the package **`java.util`**
- An **`ArrayList`** is created and named in the same way as object of any class, except that you specify the base type as follows:

```
 ArrayList<BaseType> aList =
     new ArrayList<BaseType>();
```

# Using the **`ArrayList`** Class

- An initial capacity can be specified when creating an **`ArrayList`** as well
  - The following code creates an **`ArrayList`** that stores objects of the base type **`String`** with an initial capacity of 20 items

    ```
    ArrayList<String> list =
       new ArrayList<String>(20);
    ```
  - Specifying an initial capacity does not limit the size to which an **`ArrayList`** can eventually grow
- Note that the base type of an ArrayList is specified as a *type parameter*

# Using the **`ArrayList`** Class

- The **`add`** method is used to set an element for the first time in an **`ArrayList`**

```
list.add("something");
```

  - The method name **`add`** is overloaded
  - There is also a two argument version that allows an item to be added at any currently used index position or at the first unused position

# Using the **`ArrayList`** Class

- The **`size`** method is used to find out how many indices already have elements in the **`ArrayList`**

```
int howMany = list.size();
```

- The **`set`** method is used to replace any existing element, and the **`get`** method is used to access the value of any existing element

```
list.set(index, "something else");
String thing = list.get(index);
```

# Tip: Summary of Adding to an **`ArrayList`**

- The **`add`** method is usually used to place an element in an **`ArrayList`** position for the first time (at an **`ArrayList`** index)
- The simplest **`add`** method has a single parameter for the element to be added, and adds an element at the next unused index, in order

# Tip: Summary of Adding to an `ArrayList`

- An element can be added at an already occupied list position by using the two-parameter version of `add`
- This causes the new element to be placed at the index specified, and every other member of the `ArrayList` to be moved up by one position

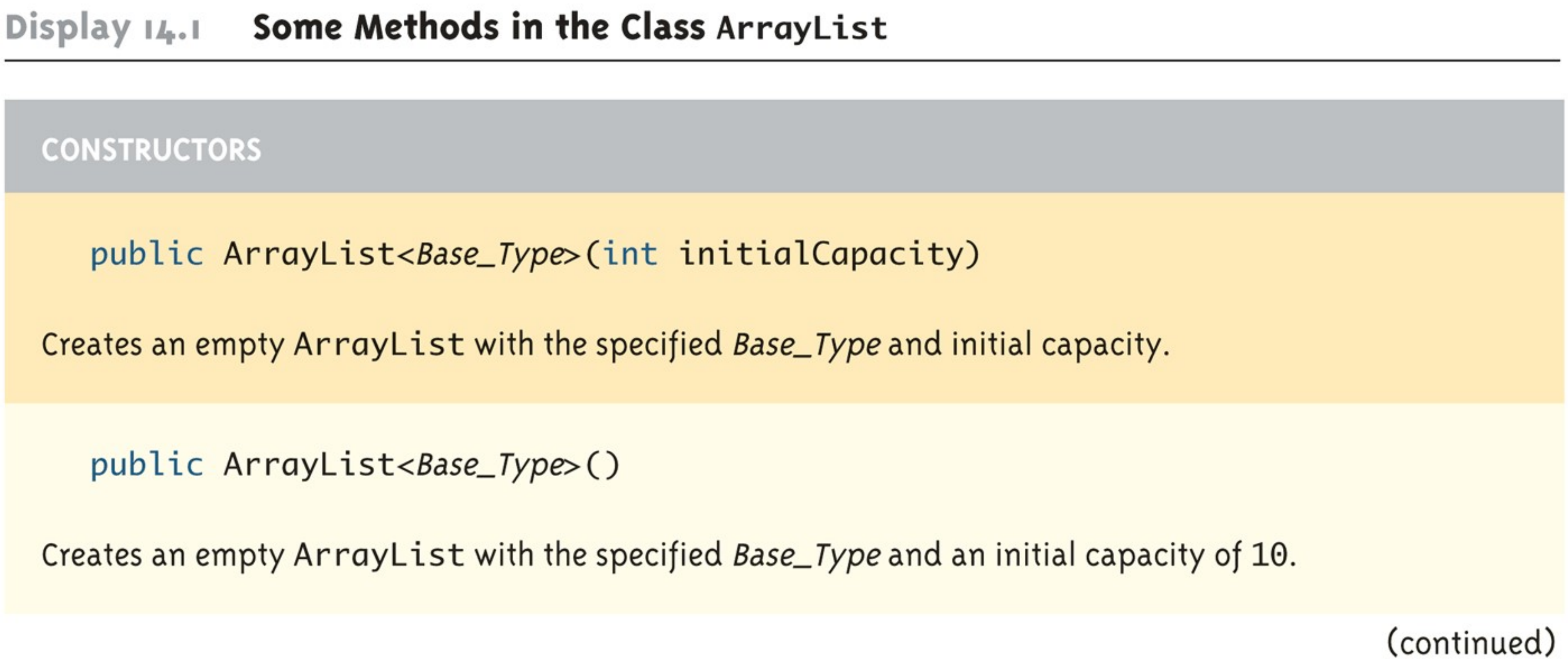
# Tip: Summary of Adding to an `ArrayList`

- The two-argument version of `add` can also be used to add an element at the first unused position (if that position is known)
- Any individual element can be changed using the `set` method
  - However, `set` can only reset an element at an index that already contains an element
- In addition, the method `size` can be used to determine how many elements are stored in an `ArrayList`

# Methods in the Class **`ArrayList`**

- The tools for manipulating arrays consist only of the square brackets and the instance variable **`length`**
- **`ArrayLists`**, however, come with a selection of powerful methods that can do many of the things for which code would have to be written in order to do them using arrays

# Some Methods in the Class **`ArrayList`** (Part 1 of 11)



Display 14.1 **Some Methods in the Class `ArrayList`**

CONSTRUCTORS

```
public ArrayList<Base_Type>(int initialCapacity)
```

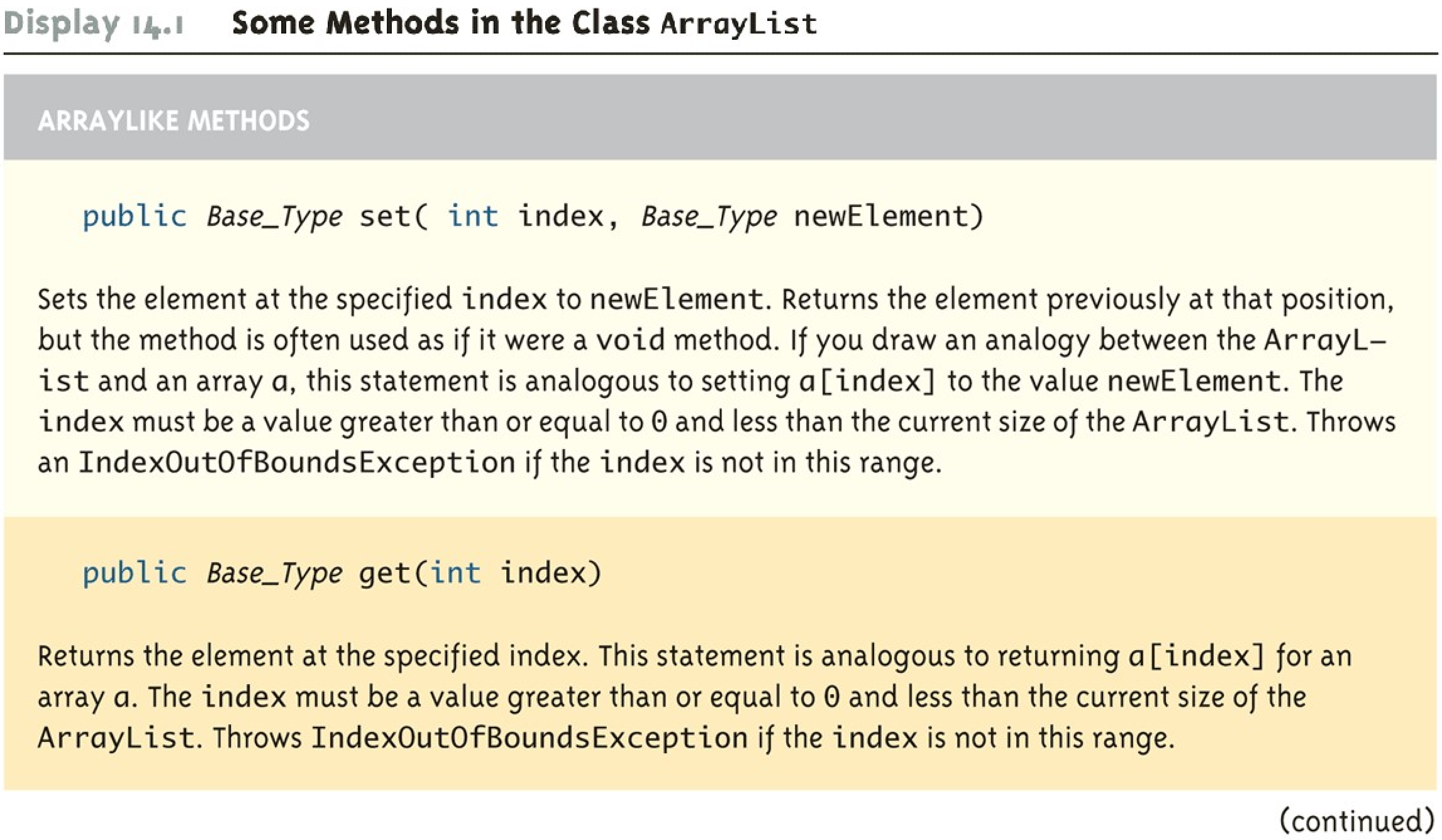
Creates an empty `ArrayList` with the specified *Base_Type* and initial capacity.

```
public ArrayList<Base_Type>()
```

Creates an empty `ArrayList` with the specified *Base_Type* and an initial capacity of `10`.

(continued)

# Some Methods in the Class **ArrayList** (Part 2 of 11)



Display 14.1 **Some Methods in the Class ArrayList**

ARRAYLIKE METHODS

```
public Base_Type set( int index, Base_Type newElement)
```

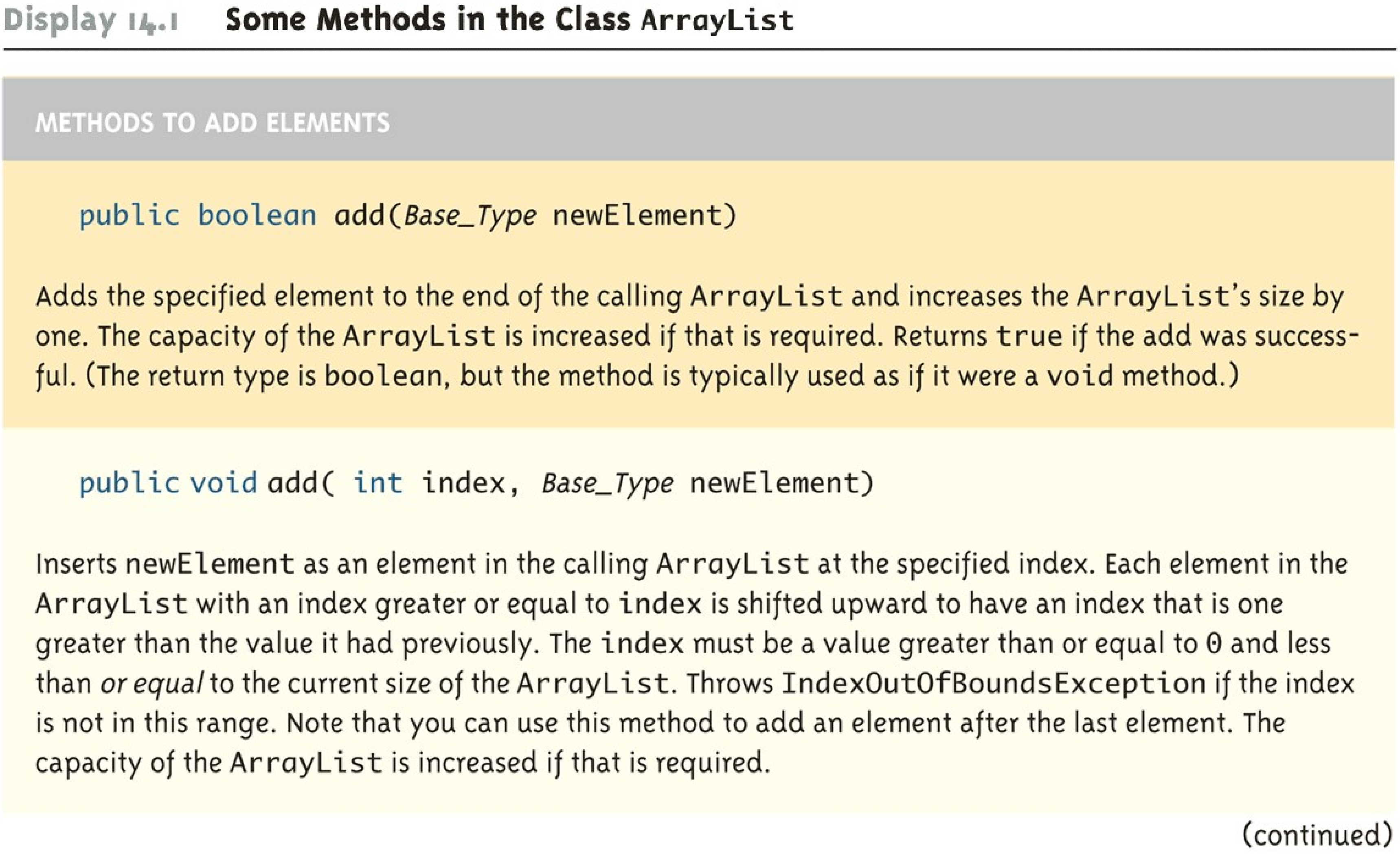
Sets the element at the specified index to newElement. Returns the element previously at that position, but the method is often used as if it were a void method. If you draw an analogy between the ArrayList and an array a, this statement is analogous to setting a[index] to the value newElement. The index must be a value greater than or equal to 0 and less than the current size of the ArrayList. Throws an IndexOutOfBoundsException if the index is not in this range.

```
public Base_Type get(int index)
```

Returns the element at the specified index. This statement is analogous to returning a[index] for an array a. The index must be a value greater than or equal to 0 and less than the current size of the ArrayList. Throws IndexOutOfBoundsException if the index is not in this range.

(continued)

# Some Methods in the Class **`ArrayList`** (Part 3 of 11)



Display 14.1 **Some Methods in the Class `ArrayList`**

**METHODS TO ADD ELEMENTS**

```
public boolean add(Base_Type newElement)
```

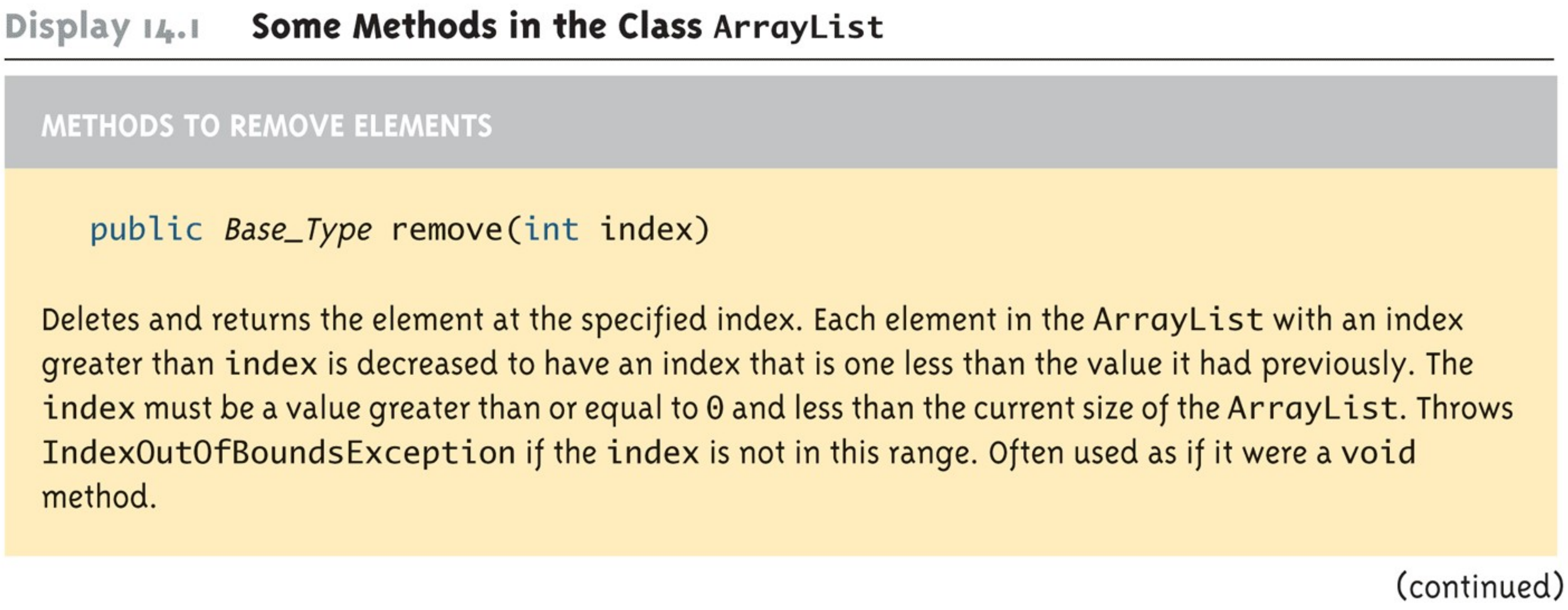
Adds the specified element to the end of the calling `ArrayList` and increases the `ArrayList`'s size by one. The capacity of the `ArrayList` is increased if that is required. Returns `true` if the add was successful. (The return type is `boolean`, but the method is typically used as if it were a `void` method.)

```
public void add( int index, Base_Type newElement)
```

Inserts `newElement` as an element in the calling `ArrayList` at the specified index. Each element in the `ArrayList` with an index greater or equal to `index` is shifted upward to have an index that is one greater than the value it had previously. The `index` must be a value greater than or equal to `0` and less than *or equal* to the current size of the `ArrayList`. Throws `IndexOutOfBoundsException` if the index is not in this range. Note that you can use this method to add an element after the last element. The capacity of the `ArrayList` is increased if that is required.

(continued)

# Some Methods in the Class `ArrayList` (Part 4 of 11)



Display 14.1 **Some Methods in the Class `ArrayList`**

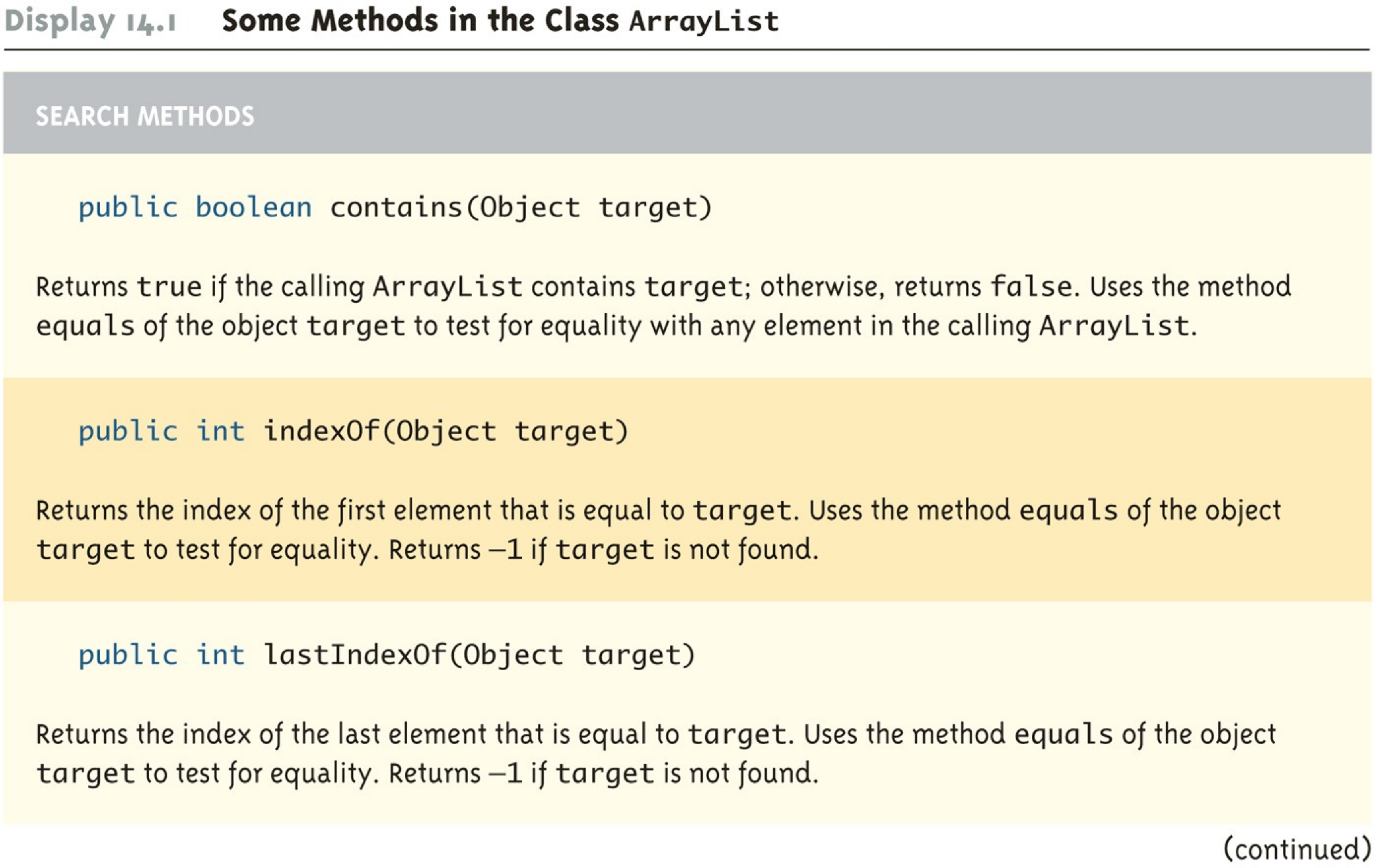
METHODS TO REMOVE ELEMENTS

```
public Base_Type remove(int index)
```

Deletes and returns the element at the specified index. Each element in the `ArrayList` with an index greater than `index` is decreased to have an index that is one less than the value it had previously. The `index` must be a value greater than or equal to `0` and less than the current size of the `ArrayList`. Throws `IndexOutOfBoundsException` if the `index` is not in this range. Often used as if it were a `void` method.

(continued)

# Some Methods in the Class **`ArrayList`** (Part 6 of 11)



Display 14.1 **Some Methods in the Class `ArrayList`**

**SEARCH METHODS**

```
public boolean contains(Object target)
```

Returns `true` if the calling `ArrayList` contains `target`; otherwise, returns `false`. Uses the method `equals` of the object `target` to test for equality with any element in the calling `ArrayList`.

```
public int indexOf(Object target)
```

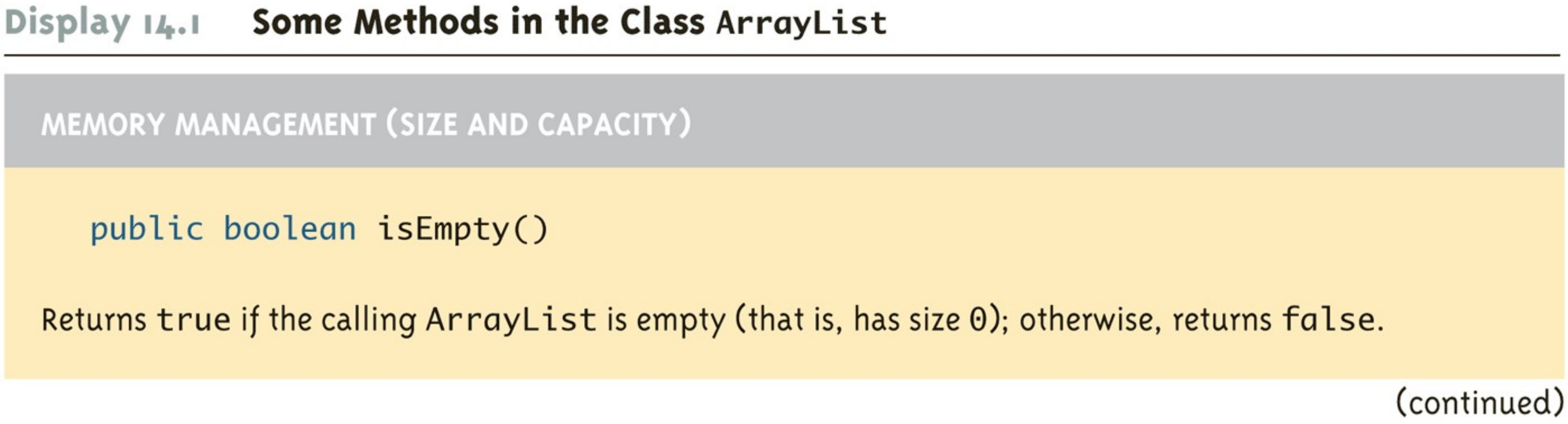
Returns the index of the first element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns –1 if `target` is not found.

```
public int lastIndexOf(Object target)
```

Returns the index of the last element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns –1 if `target` is not found.

(continued)

# Some Methods in the Class **`ArrayList`** (Part 7 of 11)



Display 14.1 **Some Methods in the Class `ArrayList`**

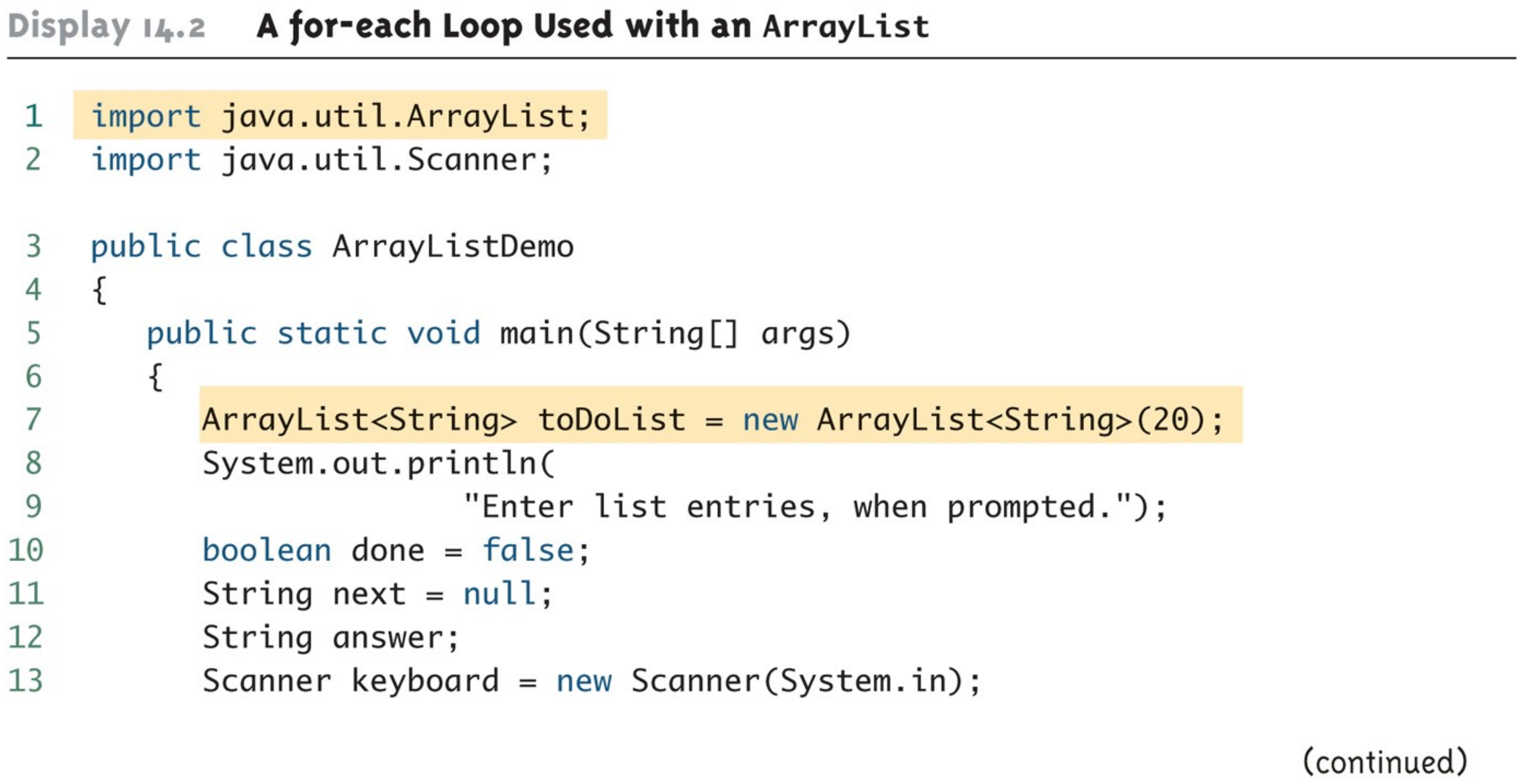
| MEMORY MANAGEMENT (SIZE AND CAPACITY) |
|---|
| `public boolean isEmpty()`<br>Returns `true` if the calling `ArrayList` is empty (that is, has size `0`); otherwise, returns `false`. |

(continued)

# The "For Each" Loop

- The `ArrayList` class is an example of a *collection* class
- Starting with version 5.0, Java has added a new kind of for loop called a *for-each* or *enhanced for* loop
  - This kind of loop has been designed to cycle through all the elements in a collection (like an `ArrayList`)

# A for-each Loop Used with an **ArrayList** (Part 1 of 3)

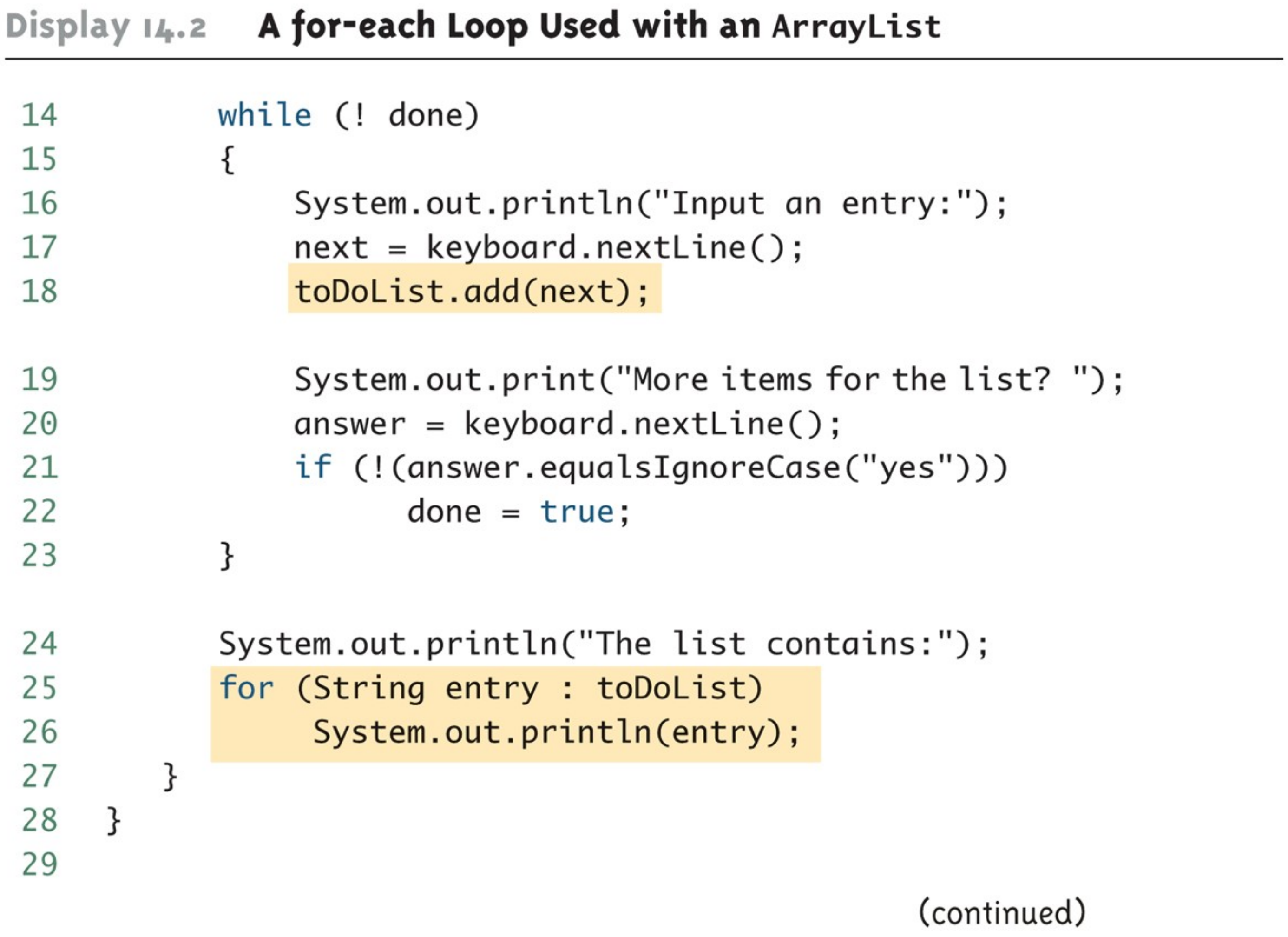


Display 14.2 **A for-each Loop Used with an ArrayList**

```
import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList<String> toDoList = new ArrayList<String>(20);
        System.out.println(
                      "Enter list entries, when prompted.");
        boolean done = false;
        String next = null;
        String answer;
        Scanner keyboard = new Scanner(System.in);
```

(continued)

# A for-each Loop Used with an **ArrayList** (Part 2 of 3)

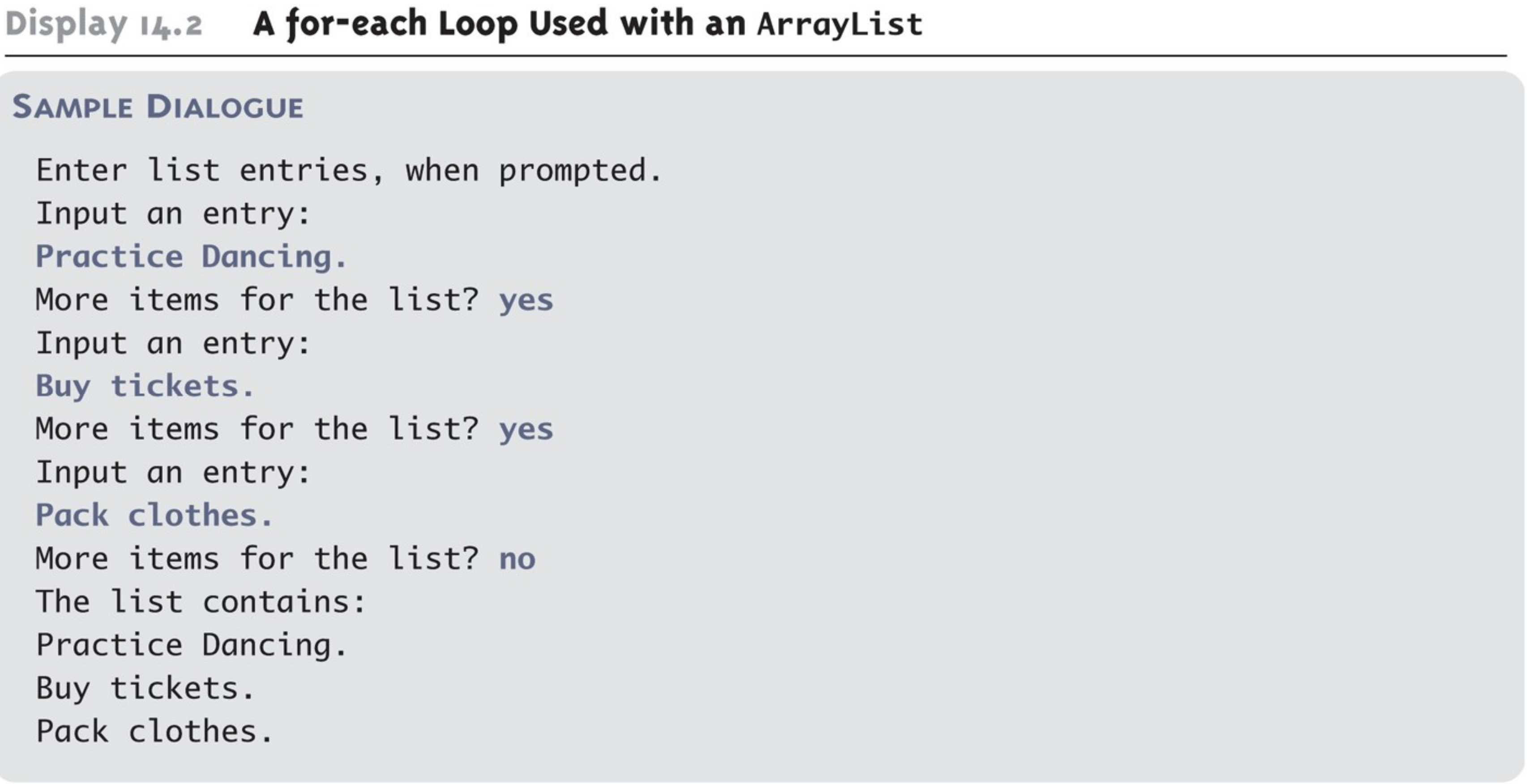


**Display 14.2 A for-each Loop Used with an ArrayList**

```
        while (! done)
        {
            System.out.println("Input an entry:");
            next = keyboard.nextLine();
            toDoList.add(next);

            System.out.print("More items for the list? ");
            answer = keyboard.nextLine();
            if (!(answer.equalsIgnoreCase("yes")))
                done = true;
        }

        System.out.println("The list contains:");
        for (String entry : toDoList)
            System.out.println(entry);
    }
}
```

(continued)

# A for-each Loop Used with an **`ArrayList`** (Part 3 of 3)



Display 14.2 **A for-each Loop Used with an `ArrayList`**

**SAMPLE DIALOGUE**

```
Enter list entries, when prompted.
Input an entry:
Practice Dancing.
More items for the list? yes
Input an entry:
Buy tickets.
More items for the list? yes
Input an entry:
Pack clothes.
More items for the list? no
The list contains:
Practice Dancing.
Buy tickets.
Pack clothes.
```

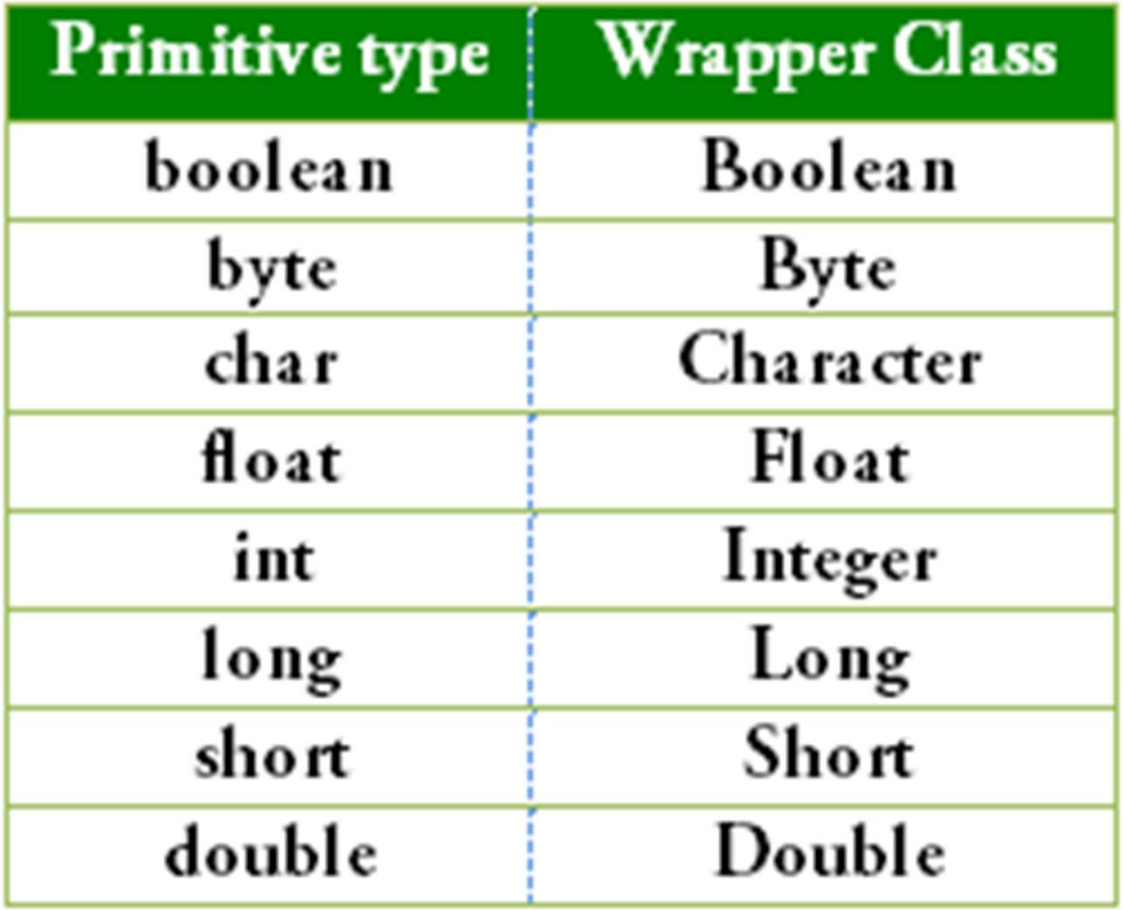
# Pitfall: The `clone` method Makes a Shallow Copy

- When a deep copy of an `ArrayList` is needed, using the clone method is not sufficient
  - Invoking `clone` on an `ArrayList` object produces a shallow copy, not a deep copy
- In order to make a deep copy, it must be possible to make a deep copy of objects of the base type
  - Then a deep copy of each element in the `ArrayList` can be created and placed into a new `ArrayList` object

# Wrapper Classes

- *Wrapper classes* provide a class type corresponding to each of the primitive types
  - This makes it possible to have class types that behave somewhat like primitive types
  - The wrapper classes for the primitive types **`byte`**, **`short`**, **`long`**, **`float`**, **`double`**, and **`char`** are (in order) **`Byte`**, **`Short`**, **`Long`**, **`Float`**, **`Double`**, and **`Character`**
- Wrapper classes also contain a number of useful predefined constants and static methods

# Wrapper Classes



| Primitive type | Wrapper Class |
|---|---|
| boolean | Boolean |
| byte | Byte |
| char | Character |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |
| double | Double |

# Wrapper Classes

- ArrayList <Integer> list = new ArrayList<Integer> (19);
- List.add(new Integer(6));
- List.add(6);
- Integer x_obj= list.get(0);
- int x = x_obj.intValue();
- int x = list.get(0);

# Wrapper Classes

- *Boxing*: the process of going from a value of a primitive type to an object of its wrapper class
  - To convert a primitive value to an "equivalent" class type value, create an object of the corresponding wrapper class using the primitive value as an argument
  - The new object will contain an instance variable that stores a copy of the primitive value
  - Unlike most other classes, a wrapper class does not have a no-argument constructor

```
Integer integerObject = new Integer(42);
```

# Wrapper Classes

- *Unboxing*: the process of going from an object of a wrapper class to the corresponding value of a primitive type
  - The methods for converting an object from the wrapper classes `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `Character` to their corresponding primitive type are (in order) `byteValue`, `shortValue`, `intValue`, `longValue`, `floatValue`, `doubleValue`, and `charValue`
  - None of these methods take an argument

```
int i = integerObject.intValue();
```

# Automatic Boxing and Unboxing

- Starting with version 5.0, Java can automatically do boxing and unboxing
- Instead of creating a wrapper class object using the **`new`** operation (as shown before), it can be done as an automatic type cast:

```
 Integer integerObject = 42;
```

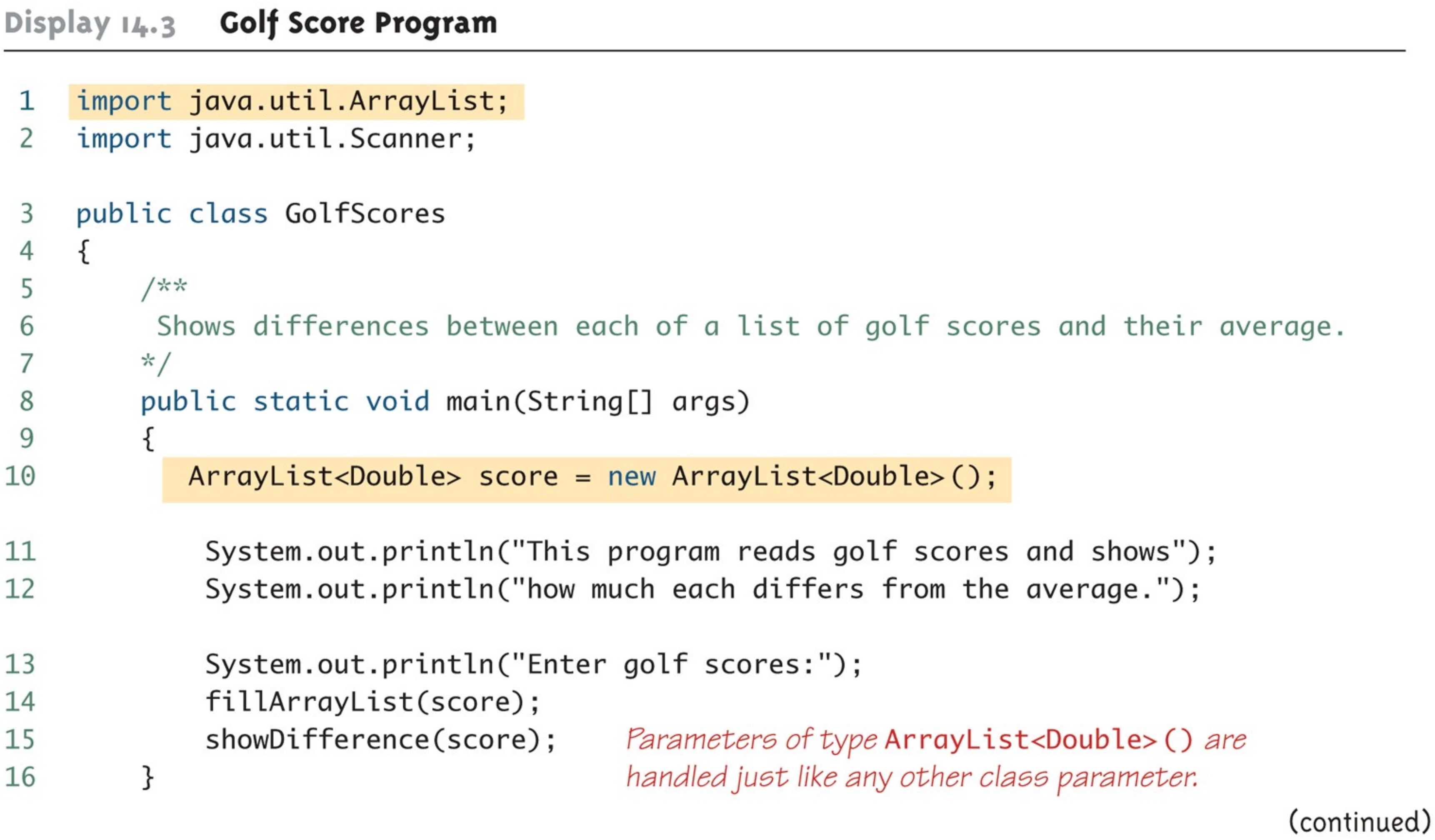
- Instead of having to invoke the appropriate method (such as **`intValue`**, **`doubleValue`**, **`charValue`**, etc.) in order to convert from an object of a wrapper class to a value of its associated primitive type, the primitive value can be recovered automatically

```
 int i = integerObject;
```

# Automatic Boxing and Unboxing

- ArrayList<Integer> array = new ArrayList<Integer> (2);
- array.add(5);
- array.add(6);
- int x = array.get(1);

# Golf Score Program (Part 1 of 6)



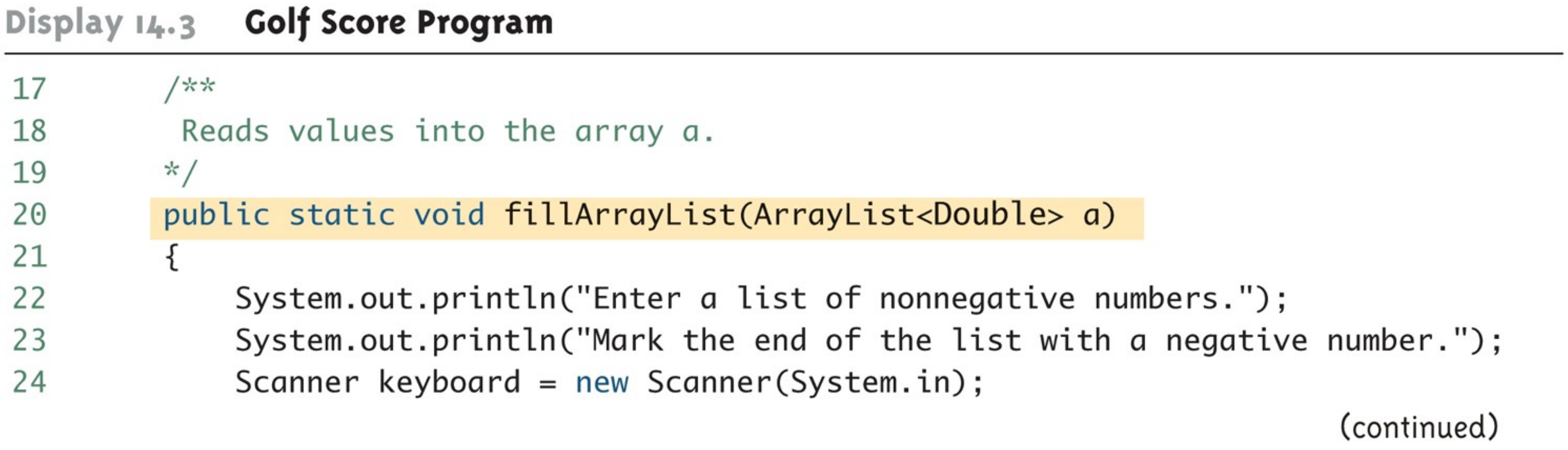
Display 14.3 **Golf Score Program**

```
import java.util.ArrayList;
import java.util.Scanner;

public class GolfScores
{
    /**
     Shows differences between each of a list of golf scores and their average.
    */
    public static void main(String[] args)
    {
       ArrayList<Double> score = new ArrayList<Double>();

        System.out.println("This program reads golf scores and shows");
        System.out.println("how much each differs from the average.");

        System.out.println("Enter golf scores:");
        fillArrayList(score);
        showDifference(score);
    }
```

*Parameters of type* `ArrayList<Double>()` *are handled just like any other class parameter.*

(continued)

# Golf Score Program (Part 2 of 6)

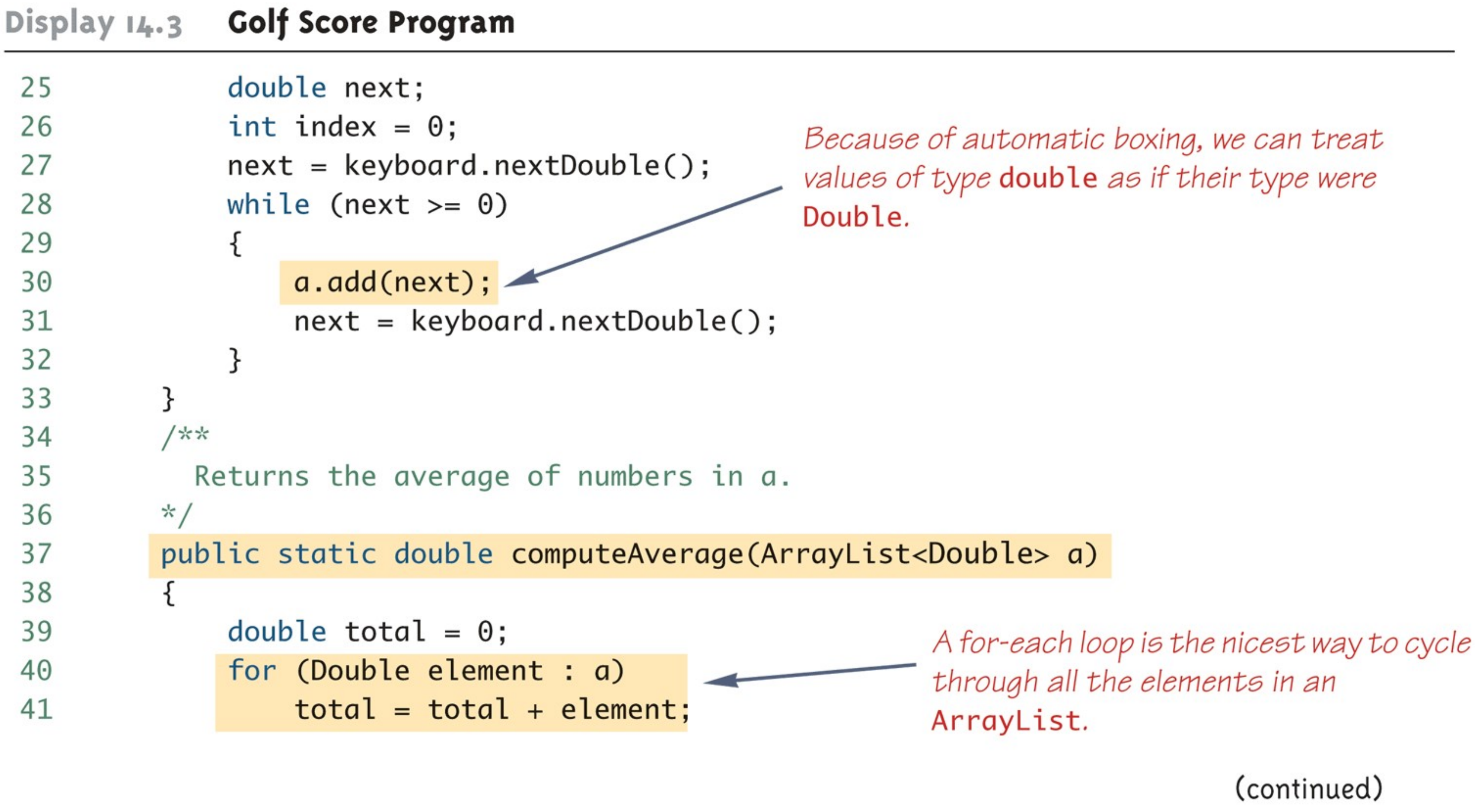


Display 14.3 **Golf Score Program**

```
    /**
     Reads values into the array a.
    */
    public static void fillArrayList(ArrayList<Double> a)
    {
        System.out.println("Enter a list of nonnegative numbers.");
        System.out.println("Mark the end of the list with a negative number.");
        Scanner keyboard = new Scanner(System.in);
```

(continued)

# Golf Score Program (Part 3 of 6)



Display 14.3 **Golf Score Program**

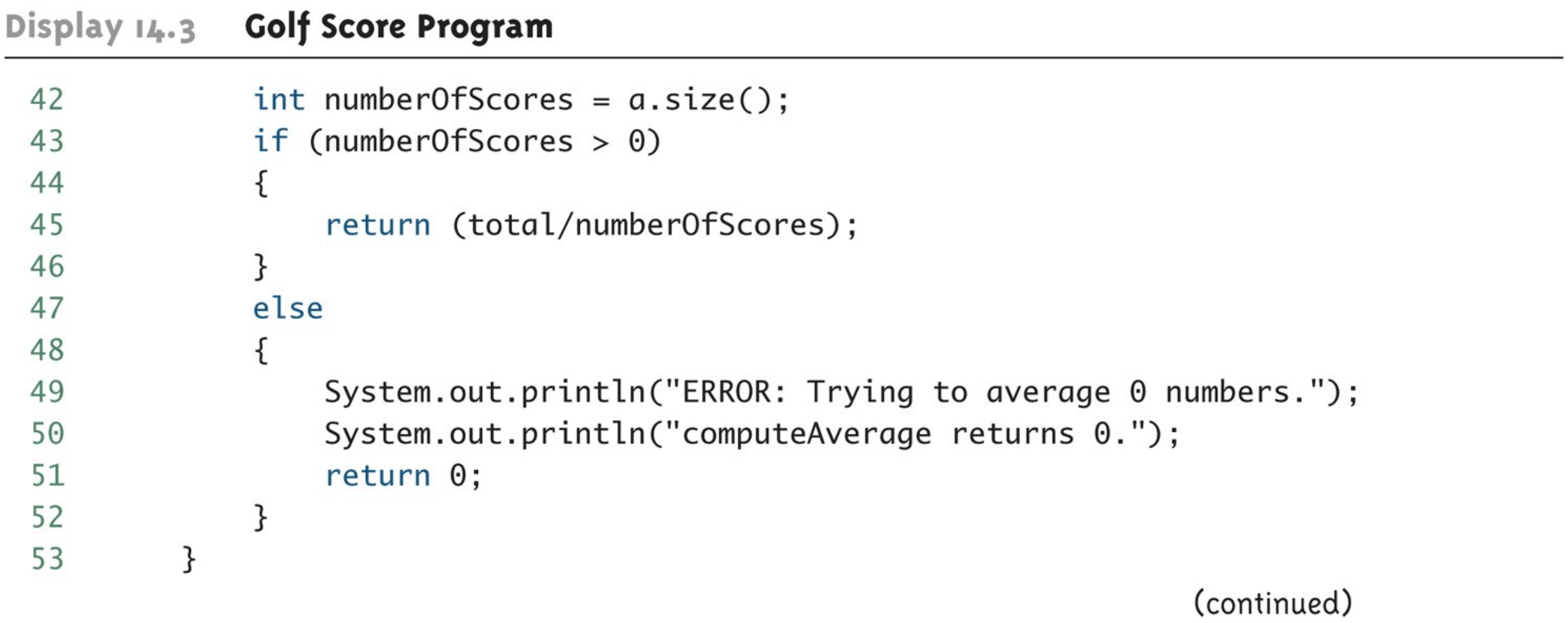
```
        double next;
        int index = 0;
        next = keyboard.nextDouble();
        while (next >= 0)
        {
            a.add(next);
            next = keyboard.nextDouble();
        }
    }
    /**
      Returns the average of numbers in a.
    */
    public static double computeAverage(ArrayList<Double> a)
    {
        double total = 0;
        for (Double element : a)
            total = total + element;
```

*Because of automatic boxing, we can treat values of type* `double` *as if their type were* `Double`.

*A for-each loop is the nicest way to cycle through all the elements in an* `ArrayList`.

(continued)

# Golf Score Program (Part 4 of 6)

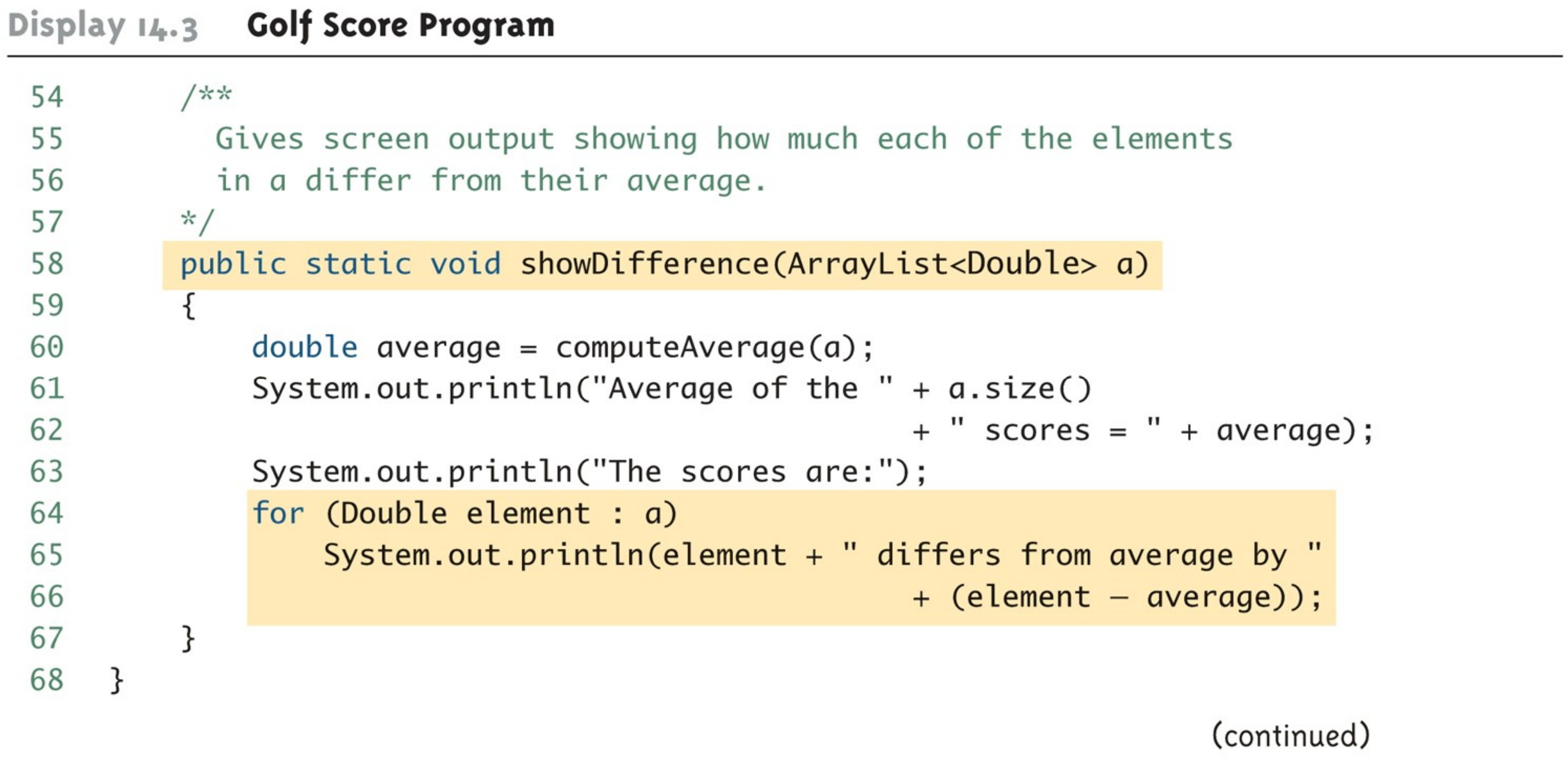


Display 14.3 **Golf Score Program**

```
        int numberOfScores = a.size();
        if (numberOfScores > 0)
        {
            return (total/numberOfScores);
        }
        else
        {
            System.out.println("ERROR: Trying to average 0 numbers.");
            System.out.println("computeAverage returns 0.");
            return 0;
        }
    }
```

(continued)

# Golf Score Program (Part 5 of 6)



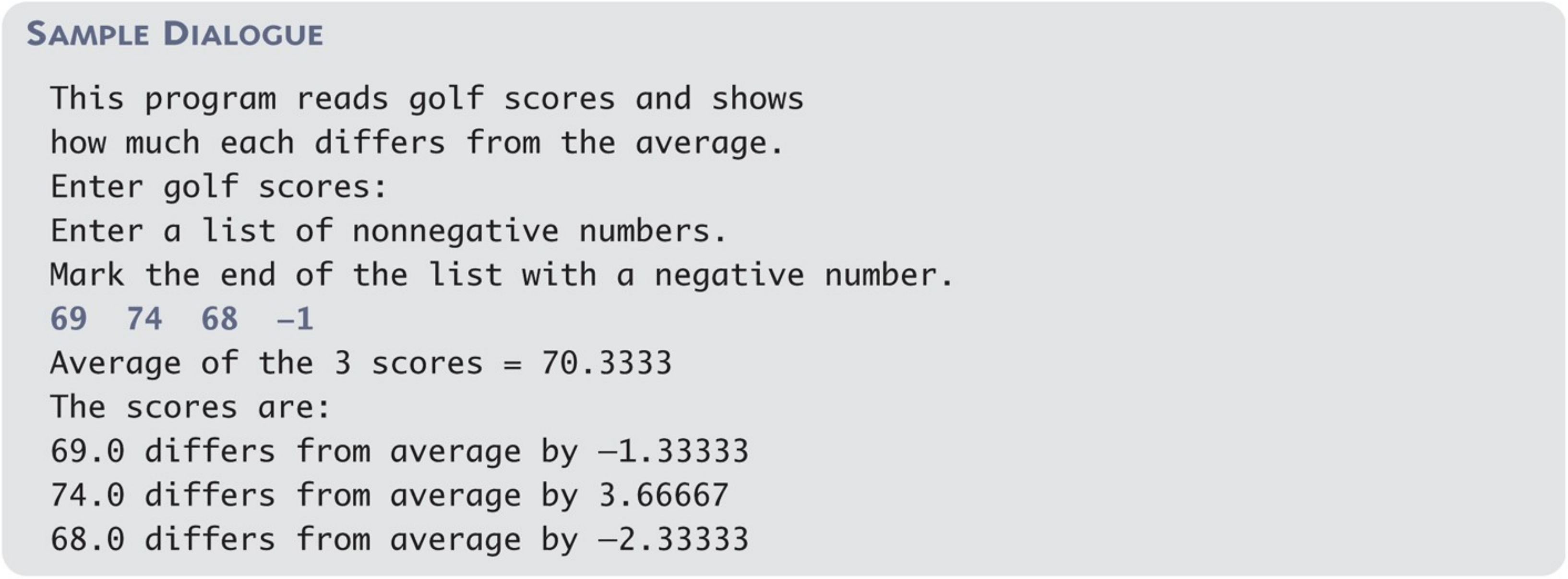
Display 14.3 **Golf Score Program**

```
    /**
      Gives screen output showing how much each of the elements
      in a differ from their average.
    */
    public static void showDifference(ArrayList<Double> a)
    {
        double average = computeAverage(a);
        System.out.println("Average of the " + a.size()
                                             + " scores = " + average);
        System.out.println("The scores are:");
        for (Double element : a)
            System.out.println(element + " differs from average by "
                                             + (element – average));
    }
}
```

(continued)

# Golf Score Program (Part 6 of 6)

Display 14.3 **Golf Score Program**



SAMPLE DIALOGUE

```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter a list of nonnegative numbers.
Mark the end of the list with a negative number.
69  74  68  –1
Average of the 3 scores = 70.3333
The scores are:
69.0 differs from average by –1.33333
74.0 differs from average by 3.66667
68.0 differs from average by –2.33333
```

# END