Java1626: Array Objects, Part 3*

Richard Baldwin

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License †

Abstract

Baldwin discusses various details regarding array objects in Java, including: members of an array object, interfaces implemented by array objects, Class objects and array objects, and the classes named Array and Arrays.

1 Table of Contents

- Preface (p. 1)
  - Viewing tip (p. 1)
    - Listings (p. 1)
- Preview (p. 2)
- Discussion and sample code (p. 2)
- Summary (p. 9)
- What's next? (p. 10)
- Miscellaneous (p. 10)
- Complete program listing (p. 11)

2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

2.1 Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

2.1.1 Listings

- Listing 1 (p. 4) . Using the newInstance method.
- Listing 2 (p. 5) . Populate the array object.
- Listing 3 (p. 5) . Display the data.
- Listing 4 (p. 6) . An array object of type int.

*Version 1.3: Dec 12, 2012 7:48 am -0600
†http://creativecommons.org/licenses/by/3.0/

- Listing 5 (p. 6) . The two-dimensional array object tree.
- Listing 6 (p. 7) . Populate the leaf elements.
- Listing 7 (p. 7) . Display the data.
- Listing 8 (p. 8) . Create, populate, and display an array object.
- Listing 9 (p. 8) . Sort and display the data.
- Listing 10 (p. 9) . Search for an existing string.
- Listing 11 (p. 9) . Search for a non-existing string.
- Listing 12 (p. 11) . Complete program listing.

3 Preview

This module discusses various details regarding the use of array objects in Java, including:

- The members of an array object
- The interfaces implemented by array objects
- **Class** objects and array objects
- The classes named **Array** and **Arrays**

4 Discussion and sample code

Members of an array object

An array object has the following members *(in addition to the data stored in the object):*

- A public final variable named **length** , which contains the number of components of the array *(length may be positive or zero)*
- A public method named **clone** . This method overrides the method of the same name in **Object** class.
- Default versions of all the methods inherited from the class named **Object** , *(other than **clone** , which is overridden as described above).*

Implements Cloneable and Serializable

Also, every array object implements the **Cloneable** and **Serializable** interfaces. *(Note that neither of these interfaces declares any methods.)*

What is the Cloneable interface?

Here is what Sun has to say about the **Cloneable** interface:

*"A class implements the **Cloneable** interface to indicate to the **Object.clone()** method that it is legal for that method to make a field-for-field copy of instances of that class. Attempts to clone instances that do not implement the **Cloneable** interface result in the exception **CloneNotSupportedException** being thrown."*

Thus, the fact than an array object implements the **Cloneable** interface makes it possible to clone array objects.

A cloned array is shallow

While it is possible to clone arrays, care must be exercised when cloning multidimensional arrays. That is because a clone of a multidimensional array is shallow.

What does shallow mean?

Shallow means that the cloning process creates only a single new array.

Subarrays are shared between the original array and the clone.

(Although I'm not certain, I suspect that this may also be the case for cloning array objects containing references to ordinary objects. I will leave that determination as an exercise for the student. In any event, be careful if you clone array objects.) **Serialization**

Serialization of an object is the process of decomposing the object into a stream of bytes, which can later be recomposed into a copy of the object. Here is what Sun has to say about the **Serializable** interface:

"Serializability of a class is enabled by the class implementing the ***java.io.Serializable*** *interface. Classes that do not implement this interface will not have any of their state serialized or deserialized.*

All subtypes of a serializable class are themselves serializable.

The serialization interface has no methods or fields and serves only to identify the semantics of being serializable."

Even though this quotation from Sun doesn't address array objects, because array objects implement the **Serializable** interface, they can be serialized and later reconstructed.

Class objects representing array objects

An object of the class named **Class** can be obtained *(by calling the* ***getClass*** *method of the* ***Object*** *class)* to represent the class from which an ordinary object was instantiated.

The **Class** object is able to answer certain questions about the class that it represents *(such as the name of the superclass),* and has other uses as well.

(One of the other uses is to specify the type as a parameter to the methods of the ***Array*** *class, which I will illustrate later in this module.)*

Every array also has an associated **Class** object.

That **Class** object is shared with all other arrays with the same component type.

The superclass of an array type is **Object** . *(Think about this!)*

An array of characters is not a string

For the benefit of the C/C++ programmers in the audience, an array of **char** is not a **String** .

(In Java, a string is an object of the ***String*** *class or the* ***StringBuffer*** *class).*

Not terminated by null

Also, neither a **String** object nor an array of type **char** is terminated by '\u0000' *(the NUL character)* .

(This information is provided for the benefit of C programmers who are accustomed to working with so-called null-terminated strings. If you're not a C programmer, don't worry about this.)

A String object in Java is immutable

Once initialized, the contents of a Java **String** object never change.

On the other hand, an array of type **char** has mutable elements. The **String** class provides a method named **toCharArray** , which returns an array of characters containing the same character sequence as a **String** .

StringBuffer objects

The class named **StringBuffer** also provides a variety of methods that work with arrays of characters. The contents of a **StringBuffer** object are mutable.

The Array and Arrays classes

The classes named **Array** and **Arrays** provide methods that you can use to work with array objects.

The **Array** class provides static methods to dynamically create and access Java arrays.

The **Arrays** class contains various methods for manipulating arrays *(such as sorting and searching).* It also contains a static factory method that allows arrays to be viewed as lists.

A sample program named Array08

The sample program named **Array08** *(shown in Listing 12 (p. 11) near the end of the module)* illustrates the use of some of these methods.

Will discuss in fragments

As usual, I will discuss this program in fragments. Essentially all of the interesting code is in the method named **main** , so I will begin my discussion there. The first few fragments will illustrate the creation, population, and display of a one-dimensional array object whose elements contain references to objects of type **String** .

The newInstance method of the Array class

The code in Listing 1 (p. 4) calls the **static** method of the **Array** class named **newInstance** to create the array object and to store the object's reference in a reference variable of type **Object** named **v1** .

*(Note that there are two overloaded versions of the **newInstance** method in the **Array** class. I will discuss the other one later.)*

Listing 1: Using the newInstance method.

```
Object v1 = Array.newInstance(Class.forName("java.lang.String"),
                              3);
```

Two parameters required

This version of the **newInstance** method requires two parameters. The first parameter specifies the component type. This must be a reference to a **Class** object representing the component type of the new array object.

The second parameter, of type **int** , specifies the length of the new array object.

The Class object

The second parameter that specifies the array length is fairly obvious. However, you may need some help with the first parameter. Here is part of what Sun has to say about a **Class** object.

*"Instances of the class **Class** represent classes and interfaces in a running Java application. Every array also belongs to a class that is reflected as a **Class** object that is shared by all arrays with the same element type and number of dimensions. The primitive Java types (boolean, byte, char, short, int, long, float, and double), and the keyword void are also represented as **Class** objects."*

Getting a reference to a Class object

I know of three ways to get *(or refer to)* a **Class** object.

- **Class** objects for primitive types
- The **getClass** method
- The **forName** method

Class objects for primitive types

There are nine predefined **Class** objects that represent the eight primitive types and void. These are created by the Java Virtual Machine, and have the same names as the primitive types that they represent: **boolean** , **byte** , **char** , **short** , **int** , **long** , **float** , and **double** . You can refer to these class objects using the following syntax:

- boolean.class,
- int.class,
- float.class, etc.

I will illustrate this later in this module.

The getClass method

If you have a reference to a target object *(ordinary object or array object),* you can gain access to a **Class** object representing the class from which that object was instantiated by calling the **getClass** method of the **Object** class, on that object.

The **getClass** method returns a reference of type **Class** that refers to a **Class** object representing the class from which the target object was instantiated.

The forName method

The static **forName** method of the **Class** class accepts the name of a class or interface as an incoming **String** parameter, and returns the **Class** object associated with the class or interface having the given string name.

*(The **forName** method cannot be used with primitive types as a parameter.)*

Class object for the String class

Referring back to Listing 1 (p. 4) , you will see that the first parameter passed to the **newInstance** method was a reference to a **Class** object representing the **String** class.

Thus, the statement in Listing 1 (p. 4) creates a one-dimensional array object, of component type **String** , three elements in length.

The reference to the array object is saved in the generic reference variable of type **Object** .

(In case you haven't recognized it already, this is an alternative to syntax such as ***new String[3]*** *.*

Note that there are no square brackets in this alternative approach. Thus, it might be said that this approach is more mainstream OOP than the approach that requires the use of square brackets.)

Populate the array object

The code in Listing 2 (p. 5) uses two **static** methods of the **Array** class to populate the three elements of the array object with references to objects of type **String** .

Listing 2: Populate the array object.

```
  for(int i = 0; i < Array.getLength(v1);i++){
 Array.set(v1, i, "a"+i);
}//end for loop
```

The getLength method

The **getLength** method of the **Array** class is used in Listing 2 (p. 5) to get the **length** of the array for use in the conditional expression of a **for** loop.

Note that unlike the sample programs in the previous module *(that stored the array object's reference as type* ***Object*** *),* it was not necessary to cast the reference to type **String[]** in order to get the **length** .

The set method

The **set** method of the **Array** class is used in Listing 2 (p. 5) to store references to **String** objects in the elements of the array object.

Again, unlike the programs in the previous module, it was not necessary to cast the array reference to type **String[]** to access the elements. In fact, there are no square brackets anywhere in Listing 2 (p. 5) .

Display the data

Listing 3 (p. 5) uses a similar **for** loop to display the contents of the **String** objects whose references are stored in the elements of the array object.

Listing 3: Display the data.

```
  for(int i = 0; i < Array.getLength(v1); i++){
 System.out.print(Array.get(v1, i)  + " ");
}//end for loop
```

No square brackets

Once again, note that no casts, and no square brackets were required in Listing 3 (p. 5) . In fact, this approach makes it possible to deal with one-dimensional array objects using a syntax that is completely devoid of square brackets.

Rather than using square brackets to access array elements, this is a *method-oriented* approach to the use of array objects. This makes it possible to treat array objects much the same as we treat ordinary objects in Java.

A two-dimensional rectangular array object tree

Next, I will use the methods of the **Array** class to create, populate, and display a rectangular two-dimensional array object tree, whose elements contain references to objects of the class **String** .

Another overloaded version of newInstance

To accomplish this, I will use the other overloaded version of the **newInstance** method. This version requires a reference to an array object of type **int** as the second parameter.

(Note that the Sun documentation describes two different behaviors for this method, depending on the whether the first parameter represents a non-array class or interface, or represents an array type. This sample program illustrates the first possibility.)

The second parameter

As mentioned above, the version of the **newInstance** method that I am going to use requires a reference to an array object of type **int** as the second parameter.

*(The length of the array object of type **int** specifies the number of dimensions of the multi-dimensional array object. The contents of the elements of the array object of type **int** specify the sizes of those dimensions.)*

Thus, my first task is to create and populate an array object of type **int** .

An array object of type int

Listing 4 (p. 6) shows the code required to create and populate the array object of type **int** . This is a one-dimensional array object having two elements *(length equals 2)*. The first element is populated with the **int** value 2 and the second element is populated with the **int** value 3.

Listing 4: An array object of type int.

```
    Object v2 = Array.newInstance(int.class,2);
Array.setInt(v2, 0, 2);
Array.setInt(v2, 1, 3);
```

Why do we need this array object?

When this array object is used later, in conjunction with the version of the **newInstance** method that requires a reference to an array object of type **int** as the second parameter, this array object will specify an array object having two dimensions *(a rectangular array)*. The rectangular array will have two rows and three columns.

Same newInstance method as before

Note that Listing 4 (p. 6) uses the same version of the **newInstance** method that was used to create the one-dimensional array object in Listing l (p. 4) .

Class object representing int

Note the syntax of the first parameter passed to the **newInstance** method in Listing 4 (p. 6) . As mentioned earlier, this is a reference to the predefined **Class** object that represents the primitive type **int** . This causes the component type of the array object to be type **int** .

The setInt method

You should also note the use of the **setInt** method of the **Array** class to populate each of the two elements in the array in Listing 4 (p. 6) *(with **int** values of 2 and 3 respectively)*.

The two-dimensional array object tree

Listing 5 (p. 6) uses the other overloaded version of the **newInstance** method to create a two-dimensional array object tree, having two rows and three columns.

Listing 5: The two-dimensional array object tree.

```
    Object v3 = Array.newInstance(Class.forName("java.lang.String"),
                                  (int[])v2);
```

A reference to the array object at the root of the tree is stored in the reference variable of type **Object** named **v3** . Note that the tree is designed to store references to objects of type **String** .

(The number of dimensions and the size of each dimension are specified by the reference to the array object of type ***int*** *passed as the second parameter.)*

Square-bracket cast is required here

The required type of the second parameter for this version of the **newInstance** method is **int[]** . Therefore, there was no way for me to avoid the use of square brackets. I could either store the reference to the array object as type **Object** and cast it before passing it to the method, *(which I did),* or save it originally as type **int[]** , *(which I didn't).* Either way, I would have to know about the type **int[]** .

Populate the leaf elements

The nested **for** loop in Listing 6 (p. 7) uses the various methods of the **Array** class to populate the elements in the leaf array objects with references to objects of the class **String** .

Listing 6: Populate the leaf elements.

```
    for(int i=0;i < Array.getLength(v3);i++){
  for(int j=0;j < Array.getLength(Array.get(v3,i));j++){
    Array.set(Array.get(v3,i),j,"b" + (i+1)*(j+1));
  }//end inner loop
}//end outer loop
```

Admittedly, the code in Listing 6 (p. 7) is a little complex. However, there is really nothing new there, so I won't discuss it further.

Display the data

Similarly, the code in Listing 7 (p. 7) uses the methods of the **Array** class in a nested **for** loop to get and display the contents of the **String** objects whose references are stored in the elements of the leaf array objects. Again, there is nothing new here, so I won't discuss this code further.

Listing 7: Display the data.

```
    for(int i=0;i < Array.getLength(v3);i++){
  for(int j=0;j < Array.getLength(Array.get(v3,i));j++){
    System.out.print(Array.get(Array.get(v3,i),j) + " ");
  }//end inner loop
  System.out.println();
}//end outer loop
System.out.println();
```

Very few square brackets

I will point out that with the exception of the requirement to create and pass an array object as type **int[]** , it was possible to write this entire example without the use of square brackets. This further illustrates the fact that the **Array** class makes it possible to create and work with array objects in a *method-oriented* manner, almost devoid of the use of square-bracket notation.

Sorting and Searching

Many college professors require their students to spend large amounts of time reinventing algorithms for sorting and searching *(and for various collections and data structures as well).* There was probably a time in history when that was an appropriate use of a student's time. However, in my opinion, that time has passed.

Reuse, don't reinvent

Through a combination of the **Arrays** class, and the **Java Collections Framework** , most of the sorting, searching, data structures, and collection needs that you might have are readily available without a requirement for you to reinvent them.

(One of the most important concepts in OOP is ***reuse, don't reinvent*** *.)*

I will now illustrate sorting and searching using **static** methods of the **Arrays** class.

*(Note that the **Arrays** class is different from the **Array** class discussed earlier.)*

Create, populate, and display an array object

To give us something to work with, Listing 8 (p. 8) creates, populates, and displays the contents of an array object. Note that the array object is populated with references to **String** objects. There is nothing new here, so I won't discuss the code in Listing 8 (p. 8) in detail.

Listing 8: Create, populate, and display an array object.

```
    Object v4 = Array.newInstance(Class.forName("java.lang.String"),
                                  8);
//Populate the array object.
// Create a gap in the data.
for(int i = 0; i < Array.getLength(v4); i++){
  if(i < 4){
    Array.set(v4,i,"c"+(8-i));}
  else{
    Array.set(v4,i,"c"+(18-i));}
}//end for loop

//Display the raw data
for(int i = 0; i < Array.getLength(v4); i++){
  System.out.print(Array.get(v4,i)+ " ");
}//end for loop
```

The output

The code in Listing 8 (p. 8) produces the following output on the screen:

```
c8 c7 c6 c5 c14 c13 c12 c11
```

Note that the order of this data is generally descending, and there is no string encapsulating the characters **c4** .

Sort and display the data

The code in Listing 9 (p. 8) uses the **sort** method of the **Arrays** class to sort the array data into ascending order.

Listing 9: Sort and display the data.

```
    Arrays.sort((Object[])v4);

//Display the sorted data
for(int i = 0; i < Array.getLength(v4); i++){
  System.out.print(Array.get(v4, i) + " ");
}//end for loop
```

The output

The code in Listing 9 (p. 8) displays the sorted contents of the array object, producing the following **output on the computer screen** :

```
c11 c12 c13 c14 c5 c6 c7 c8
```

Note that the order of the data in the array object has been modified, and the array data is now in ascending order.

(This order is based on the natural ordering of the ***String*** *data. I discuss other ways to order sorted data in conjunction with the* ***Comparable*** *and* ***Comparator*** *interfaces in my modules on the Java Collections Framework.)*

Binary search

A binary search is a search algorithm that can very quickly find an item stored in a sorted collection of items. In this case, the collection of items is stored in an array object, and the data is sorted in ascending order.

Search for an existing string

Listing 10 (p. 9) uses the **binarySearch** method of the **Arrays** class to perform a search for an existing **String** object whose reference is stored in the sorted array. The code searches for the reference to the **String** object encapsulating the characters **c5** .

Listing 10: Search for an existing string.

```
System.out.println(Arrays.binarySearch((Object[])v4,"c5"));
```

The result of the search

The code in Listing 10 (p. 9) displays the numeral 4 on the screen.

When the **binarySearch** method finds a match, it returns the index value of the matching element. If you go back and look at the sorted contents (p. 8) of the array shown earlier, you will see that this is the index of the element containing a reference to a **String** object that encapsulates the characters **c5** .

Search for a non-existing string

The code in Listing 11 (p. 9) uses the **binarySearch** method to search for a reference to a **String** object that encapsulates the characters **c4** . As I indicated earlier, a **String** object that encapsulates these characters is not represented in the sorted array object.

Listing 11: Search for a non-existing string.

```
System.out.println(Arrays.binarySearch((Object[])v4,"c4"));
```

The result of the search

The code in Listing 11 (p. 9) produces the following negative numeral on the screen: **-5** .

Here is Sun's explanation for the value returned by the **binarySearch** method:

"Returns: index of the search key, if it is contained in the list; otherwise, (-(insertion point) - 1). The insertion point is defined as the point at which the key would be inserted into the list: the index of the first element greater than the key, or list.size(), if all elements in the list are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found."

Thus, the negative return value indicates that the method didn't find a match. The absolute value of the return value can be used to determine the index of the reference to the target object if it did exist in the sorted list. I will leave it as an exercise for the student to interpret Sun's explanation beyond this simple explanation.

Other capabilities

In addition to sorting and searching, the **Arrays** class provides several other methods that can be used to manipulate the contents of array objects in Java.

5 Summary

An array object has the following members *(in addition to the data stored in the object):*

- A public final variable named **length**

- An overridden version of the public method named **clone**
- Default versions of all the other methods inherited from the class named **Object**

Every array object implements the **Cloneable** and **Serializable** interfaces.

A clone of a multidimensional array is shallow. Therefore, you should exercise caution when cloning array objects.

Because array objects implement the **Serializable** interface, they can be serialized and later reconstructed.

Every array also has an associated **Class** object.

The classes named **Array** and **Arrays** provide methods that you can use to work with array objects.

The **Array** class provides static methods to dynamically create and access Java array objects.

The **Arrays** class contains various methods for manipulating arrays *(such as sorting and searching)*. It also contains a static factory method that allows arrays to be viewed as lists.

Class objects are required when using the methods of the **Array** class to dynamically create Java array objects.

There are nine predefined **Class** objects that represent the eight primitive types and void. They are accessed using the following syntax: boolean.class, int.class, etc.

Three ways to get a **Class** object are:

- Class objects for primitive types: **int.class** , etc.
- The **getClass** method
- The **forName** method

The methods of the **Array** class make it possible to deal with one-dimensional array objects using a syntax that is completely devoid of square brackets. This is a *method-oriented* approach to the use of array objects. This makes it possible to treat array objects much the same as we treat ordinary objects in Java. The required syntax for multi-dimensional array objects is mostly devoid of square brackets.

The **Arrays** class provides methods for sorting and searching array objects as well as performing other operations on array objects as well.

Through a combination of the **Arrays** class and the Java Collections Framework, most of the sorting, searching, data structures, and collection needs that you might have are readily available without a requirement for you to reinvent them.

One of the most important concepts in OOP is *reuse, don't reinvent.*

6 What's next?

The next module will explain the use of the ***this*** and ***super*** keywords.

7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Array Objects, Part 3
- File: Java1626.htm
- Published: August 8, 2012
- Revised: –

NOTE: **Disclaimers: Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a

pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

8 Complete program listing

A complete listing of the program is shown in Listing 12 (p. 11) below.

Listing 12: Complete program listing.

```
    /*File Array08.java
Copyright 2002, R.G.Baldwin
Rev 2/10/02

This program illustrates the use of
static methods of the Array class to
dynamically create and access Java
arrays.

It also illustrates the use of static
methods of the Arrays class to sort
and search array objects.

Tested using JDK 1.3 under Win 2000.
*************************************/
import java.lang.reflect.Array;
import java.util.Arrays;

public class Array08{
  public static void main(
                        String[] args){

  try{
    //Create, populate, and display a
    // one-dimensional array object
    // whose elements contain
    // references to objects of type
    // String.

    //Create the array object
    Object v1 = Array.newInstance(
```

```
          Class.forName(
           "java.lang.String"), 3);
//Populate the array object
for(int i = 0; i <
         Array.getLength(v1); i++){
  Array.set(v1, i, "a"+i);
}//end for loop
//Display the data
for(int i = 0; i <
         Array.getLength(v1); i++){
  System.out.print(
           Array.get(v1, i) + " ");
}//end for loop
System.out.println();
System.out.println();

//Create, populate, and display a
// rectangular two-dimensional
// array object tree whose
// elements contain references
// to objects of type String.

//First create an array object of
// type int required as a
// parameter to the newInstance
// method.  Populate it to later
// specify a rectangular array
// object tree with two rows and
// three columns.
Object v2 = Array.newInstance(
                     int.class, 2);
Array.setInt(v2, 0, 2);
Array.setInt(v2, 1, 3);
//Now create the actual two-
// dimensional array object tree.
Object v3 = Array.newInstance(
   Class.forName(
   "java.lang.String"), (int[])v2);

//Populate the leaf elements with
// references to objects of type
// String.
for(int i=0;i<

          Array.getLength(v3);i++){
  for(int j=0;j<
           Array.getLength(
             Array.get(v3,i));j++){
    Array.set(Array.get(v3,i), j,
                "b" + (i+1)*(j+1));
  }//end inner loop
```

```
}//end outer loop

//Display the data encapsulated
// in the String objects.
for(int i=0;i<Array.getLength(v3);
                              i++){
  for(int j=0;j<Array.getLength(
             Array.get(v3,i));j++){
    System.out.print(Array.get(
        Array.get(v3,i), j) + " ");
  }//end inner loop
  System.out.println();
}//end outer loop
System.out.println();

//Now illustrate sorting and
// searching using methods of
// the arrays class.

//Create the array object
Object v4 = Array.newInstance(
          Class.forName(
           "java.lang.String"), 8);
//Populate the array object.
// Create a gap in the data.
for(int i = 0; i <
         Array.getLength(v4); i++){
  if(i<4){Array.set(v4, i,
                       "c"+(8-i));}
  else{Array.set(v4, i,
                      "c"+(18-i));}
}//end for loop

//Display the raw data
for(int i = 0; i <
         Array.getLength(v4); i++){
  System.out.print(Array.get(v4, i)
                             + " ");
}//end for loop
System.out.println();

//Sort array data into
// ascending order.
Arrays.sort((Object[])v4);

//Display the sorted data
for(int i = 0; i <
         Array.getLength(v4); i++){
  System.out.print(
           Array.get(v4, i) + " ");
}//end for loop
```

```
    System.out.println();

    //Search for an existing String
    System.out.println(
      Arrays.binarySearch((Object[])v4,
                                "c5"));
    //Search for a non-existing String
    System.out.println(
      Arrays.binarySearch((Object[])v4,
                                "c4"));

  }catch(ClassNotFoundException e){
                System.out.println(e);}
  }//end main
}//end class Array08
```

-end-