Java1622: Array Objects, Part 1*

R.G. (Dick) Baldwin

Abstract

Baldwin shows how array objects fit into the grand scheme of things in OOP using Java.

1 Table of Contents

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

*Version 1.2: Dec 12, 2012 7:48 am -0600
†http://creativecommons.org/licenses/by/3.0/

3 Preview

This module explains how array objects fit into the grand scheme of things in Object-Oriented Programming *(OOP)* using Java.

A different syntax is required to create array objects than the syntax normally used to create ordinary objects.

Array objects are accessed via references. Any of the methods of the **Object** class can be called on a reference to an array object.

Array objects encapsulate a group of variables. The variables don't have individual names. They are accessed using positive integer index values. The integer indices of a Java array object always extend from **0** to **(n-1)** where **n** is the **length** of the array encapsulated in the object.

All array objects in Java encapsulate one-dimensional arrays. The component type of an array may itself be an array type. This makes it possible to create array objects whose individual components refer to other array objects. This is the mechanism for creating *multi-dimensional* or *ragged* arrays in Java.

4 Discussion and sample code

Three kinds of objects

In an earlier module, I told you that from a conceptual viewpoint, there are at least three kinds of objects involved in a Java program:

- Ordinary objects
- Class objects
- Array objects

Ordinary objects

Most of the discussion up to that point in the collection dealt with what I have referred to in the above list as *ordinary objects* .

These are the objects that you instantiate in you code by applying the **new** operator to a constructor for a class in order to create a new instance *(object)* of that class.

Class objects

In that module, I emphasized that my discussion of **Class** objects was conceptual in nature and did not necessarily represent an actual implementation. I went on to discuss the class named **Class** , and discussed how the use of that class fits into the grand scheme of OOP in Java. I explained how the existence of *class variables* and *class methods* tends to complicate the rather simple OOP structure consisting only of ordinary objects.

Array objects

I haven't discussed *array objects* up to this point in this collection. That is the purpose of this module.

Also tends to complicate

The existence of array objects also tends to complicate the OOP structure of a Java program consisting only of ordinary objects. Even if you don't consider array objects to be a different kind of object, you must at least consider them to be a *special* kind of object. A completely different syntax is required to create array objects than the syntax normally used to instantiate ordinary objects.

References to array objects

Arrays are objects in Java *(at least, arrays are always encapsulated in objects).* Array objects are dynamically created. Like ordinary objects, array objects are accessed via references. The reference to an array object may be assigned to a reference variable whose type is specified as:

```
TypeName[]
```

For example, Listing 1 (p. 3) shows some unrelated declarations for variables that are capable of storing references to array objects.

Listing 1: Sample variable declarations for array objects.

```
int[] x1;

Button[] x2;

Object[] x3;
```

Note the empty square brackets that are required in the variable declarations in Listing 1 (p. 3) .

The special case of type Object

In addition, a reference to an array object may be assigned to a reference variable of type **Object** as shown in Listing 2 (p. 3) .

Listing 2: The special case of type Object.

```
Object x4;
```

Note that there are *no square brackets* in the statement in Listing 2.

What does this mean?

This means that like ordinary objects, a reference to an array object can be treated as type **Object** *(with no square brackets).*

This further means that any of the methods defined in the **Object** class *(such as the* ***toString*** *and* ***getClass*** *methods)* can be called on a reference to an array object.

The String representation of an array object's reference

For example, when the **toString** method is called on a reference to an array object containing data of type **int** , the resulting string will be similar to the following:

```
[I@73d6a5
```

Pretty ugly, huh?

You may recognize this as being similar to the default **String** returned by calling the **toString** method on an ordinary object with the name of the class for the ordinary object being replaced by **[I** .

For example, the **String** returned by calling the **toString** method on an object of the class named **Array04** , *(with no overridden toString method),* looks something like the following.

```
Array04@73d6a5
```

(Note that the hexadecimal numeric values following the @ in both of the above examples will change from one case to the next.)

Calling the getClass method

Similarly, calling the **getClass** method on references to arrays containing data of the types **Array04** , **Button** , and **int** , respectively, and then calling the **toString** method on the **Class** objects returned by the **getClass** method, produces the following:

```
class [LArray04;
class [Ljava.awt.Button;
class [I
```

Complicating the OOP structure

I made the following statement in an earlier paragraph:

"The existence of array objects also tends to complicate the OOP structure of a Java program consisting only of ordinary objects."

Array object is not a subclass of class Object

An array object can be treated as type **Object** for purposes of calling the methods of the **Object** class on the reference to the array object. However, it would probably be misleading to say that an array object is instantiated from a subclass of the **Object** class.

The new operator and the constructor name

Ordinary objects are created by applying the **new** operator to the constructor for a class, where the name of the constructor is always the same as the name of the class. That is not the case with array objects. Array objects are created by applying the **new** operator to the name of the type of data to be encapsulated in the array object.

Passing parameters versus square-bracket notation

In addition, whereas the instantiation of ordinary objects involves parameters passed in parentheses, a square-bracket notation is used instead of parentheses to create an array object. The value in the square brackets specifies the **length** of the array.

Creating an array object

Array objects *(with default initialization values)* are created by applying the **new** operator to the name of the data type to be stored in the array, using a square-bracket notation. An example is shown by the right-hand portion of the first statement in Listing 3 (p. 4) .

Listing 3: Creating array objects.

```
int[] x1 = new int[5];

int[] x2 = {1,2,3,4,5};
```

A five-element array object

The first statement in Listing 3 (p. 4) creates an array object capable of storing five values of type **int** . The statement also assigns the array object's reference to the newly-declared reference variable named **x1** .

Default initial values

Each element in the array is initialized to the default value zero.

(All array elements created in this manner receive a default initial value. Numeric primitive types receive an initial value of zero. Elements of type ***boolean*** *receive an initial value of* ***false*** *. Elements whose type is the name of a class or the name of an interface receive an initial value of* ***null*** *.)*

Explicit initialization

The second statement in Listing 3 (p. 4) also creates an array object capable of storing five values of type **int** , but in this case, the values in the elements are explicitly initialized to the values shown.

(Note that the ***new*** *operator is not used in the second statement in Listing 3. This is also a significant departure from the syntax used to instantiate ordinary objects.)*

This array object's reference is assigned to the reference variable named **x2** .

Note the empty square brackets in the variable declarations

The syntax of the type specification for the reference variable in each statement in Listing 3 (p. 4) is different from the syntax used in the type specification for either a primitive variable or an ordinary class type reference variable *(note the square brackets on the left in Listing 3)* . In Listing 3 (p. 4) , the type specifications indicate that each variable is capable of holding a reference to an array object.

The size of the array

Furthermore, the empty square brackets *(in the declaration of the reference variable)* indicate that the reference variable doesn't know *(and doesn't care)* about the size of the array to which it may refer. Each

of the reference variables declared in Listing 3 (p. 4) can refer to a one-dimensional array object of any size. Also, each of the reference variables can refer to different array objects at different points in time during the execution of the program.

> NOTE: **The Array class** As an aside, let me mention that there is a class named **Array** , which provides **static** methods to dynamically create and access Java arrays. The use of the methods of this class makes it possible to handle arrays with a programming style similar to the programming style typically used with ordinary objects. However, the use of the methods of the **Array** class tends to require more programming effort than the square-bracket notation discussed in this module. I will discuss a sample program that illustrates the methods of the **Array** class in a future module.

Encapsulating a group of variables

As is the case with other languages that support arrays, array objects in Java encapsulate a group of variables.

Zero or more variables may be encapsulated in an array object. If the number is zero, the array object is said to be empty.

(An example of an empty array object is the ***String[]*** *array passed to the* ***main*** *method in a Java application when the user doesn't enter any arguments at the command line.)*

No individual names

Also, as with other languages that support arrays, the variables encapsulated in an array object don't have individual names. Rather, they are referenced using positive integer index values.

(Typically, in Java, the index is placed in square brackets, which are applied to the name of the reference variable holding a reference to the array object.)

Elements or components?

It is common in the literature to refer to the variables that make up an array as its *elements* . However, the Java specification refers to them as *components.* The specification ascribes a different meaning to the word element, as shown in the following quotation from the specification:

"The value of an array component of type ***float*** *is always an element of the* ***float*** *value set ...; similarly, the value of an array component of type* ***double*** *is always an element of the* ***double*** *value set."*

Another quotation from Sun *(shown later in this module)* provides a somewhat clearer distinction between the words *component* and *element* .

(However, from force of habit, I will probability use component and element interchangeably in this module.)

The length of an array

If an array has **n** components, the **length** of the array is **n** . The components of the array are referenced using integer indices from 0 to (n - 1), inclusive.

Another quotation from Sun

Here is another quotation from the Java specification that explains the type specifications for the variable declarations in Listing 1 (p. 3) and Listing 3 (p. 4) .

"All the components of an array have the same type, called the component type of the array. If the component type of an array is T, then the type of the array itself is written T[]."

Components may be of an array type

As of the time that this object was originally written, all array objects in Java encapsulate one-dimensional arrays *(I have read that this may change in the future).*

The component type of an array may itself be an array type. This makes it possible to create array objects whose individual components refer to other array objects.

Multi-dimensional or ragged arrays

One way to think of this is to think of the second level of array objects as being sub-arrays of the original array object. This construct can be used to create multi-dimensional array structures.

(The geometry of such multi-dimensional array structures is not constrained to be rectangles, cubes, etc., as is the requirement in many other languages. Some authors may refer to this as ragged arrays.)

Tree structures

This process of having the components of an array contain references to sub-arrays can be continued indefinitely *(well, maybe not indefinitely, but further than I care to contemplate).*

(This can be thought of as a tree structure where each array object containing references to other array objects is a node in the tree.)

The leaves of the tree

Eventually, the components *(the leaves of the tree structure)* must refer to a component type that is not an array type. According to Sun:

"... this is called the element type of the original array, and the ***components*** *at this level of the data structure are called the* ***elements*** *of the original array."*

Component versus element

Hopefully, the above quotation provides a somewhat clearer distinction between the use of the words *component* and *element* than was presented earlier.

Generic references

The reference to any array object can also be assigned to reference variables of the types **Object** , **Cloneable** , or **Serializable** .

(***Object*** *is the class at the top of the inheritance hierarchy.* ***Cloneable*** *and* ***Serializable*** *are interfaces, which are implemented by all array objects. Thus, a reference to an array object can be treated as any of these three types.)*

Generic array objects

Therefore, if the element type of an array object is one of these types, the elements in the array can refer to:

- Other array objects
- Ordinary objects
- A mixture of the two

This is illustrated in the sample program named **Array05** shown in Listing 9 (p. 11) near the end of the module.

Will explain in fragments

I will explain this program in fragments. Listing 4 (p. 6) shows the beginning of the controlling class and the beginning of the **main** method for the program named **Array05** ..

Listing 4: The beginning of the class named Array05.

```
public class Array05{
  public static void main(String[] args){
    int[] v1 = {1,2,3,4,5};
    Object[] v2 = new Object[2];
```

Listing 4 (p. 6) creates two array objects.

An array of type int

The first array object is a five-element array of element type **int** , with the element values initialized as shown by the values within the curly brackets. The reference to this array object is assigned to the reference variable named **v1** .

An array of element type Object

The second array object is a two-element array of element type **Object** , with each of the element values initialized to their default value of **null** . The reference to the array object is assigned to the reference variable named **v2** .

(Note that unlike the previous discussion of ***Object*** *, the declaration of the reference variable in this case does include empty square brackets. I will have more to say about this later.)*

A new object of this class

Listing 5 (p. 7) creates a new *ordinary object* of class **Array05** . The code assigns the object's reference to the first element in the array object of element type **Object** , referred to by the reference variable named **v2** .

Listing 5: A new ordinary object of class Array05.

```
v2[0] = new Array05();
```

This is allowable because the reference to an object of any class can be assigned to a reference variable of type **Object** .

(The array object referred to by ***v2*** *contains two elements, each of which is a reference variable of type* ***Object*** *.)*

Populate the second element

The code in Listing 6 (p. 7) assigns the reference to the existing array object of the element type **int** to the second element in the array object of element type **Object** .

Listing 6: Populate the second element.

```
v2[1] = v1;
```

This is allowable because a reference to any array object can be assigned to a reference variable of type **Object** .

Array contains two references

At this point, the array object of element type **Object** contains two references.

(Each of the elements in an array of the declared type ***Object[]*** *is a reference of type* ***Object*** *.)*

The first element refers to an ordinary object of the class **Array05** .

The second element refers to an array object of type **int** , having five elements, populated with the integer values of 1 through 5 inclusive.

(Note that this is not a multi-dimensional array in the traditional sense. I will discuss the Java approach to such multi-dimensional arrays in the next module. This is simply a generic array of element type ***Object*** *, one element of which happens to contain a reference to an array object of type* ***int*** *.)*

Print some data

The code in Listing 7 (p. 7) passes each of the references to the **println** method of the **PrintStream** class.

Listing 7: Print some data.

```
System.out.println(v2[0]);
System.out.println(v2[1]);
```

The **println** method causes the **toString** method to be called on each reference. The **String** returned by the **toString** method is displayed on the computer screen in each case.

This is allowable because any method defined in the **Object** class *(including the* ***toString*** *method)* can be called on any reference stored in a reference variable of type **Object** .

This is true regardless of whether that reference is a reference to an ordinary object or a reference to an array object.

The output

Listing 7 (p. 7) causes the following two lines of text to be displayed:

```
Array05@73d6a5
[I@111f71
```

Pretty ugly, huh?

In both cases, this is the value of the **String** returned by the default version of the **toString** method defined in the **Object** class. Here is what Sun has to say about that default behavior:

*"Returns a string representation of the object. In general, the **toString** method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.*

*The **toString** method for class **Object** returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object."*

Doesn't address array objects

Obviously, this description of behavior doesn't address the case where the object is an array object, unless the characters **[I** are considered to be the name of a class. *(I will have a little more to say about this later.)*

Produce some more output

Finally, Listing 8 (p. 8) shows the last statement in this simple program.

Listing 8: Produce some more output.

```
    System.out.println( ( (int[])v2[1] )[4] );
  }//end main
}//end class Array05
```

What does this mean?

As you can see, the syntax of this statement is pretty ugly.

Values are accessed from an array object by following the array's reference with a pair of square brackets containing an integer index value as follows:

```
v2[1]
```

Get the value at index 1 as type Object

This code begins by accessing the component at index value 1 of the array object referred to by the reference variable named **v2** .

The value retrieved is a reference, and is retrieved as type **Object** , *(because the variable named **v2** was declared to be of type **Object[]**).*

A cast is required

A cast is used to convert from type **Object[]** to type **int[]** using the following code:

```
(int[])
```

This produces a reference to an array object capable of containing values of type **int** .

Apply index to the int array

After the type of the reference has been converted, the accessor **[4]** is applied to the reference. This causes the **int** value stored in the array object of type **int** *(at index value 4)* to be returned.

(If you refer back to Listing 4 (p. 6) , you will see that the integer value 5 was stored in the element at index value 4 of this array object.)

You should try to remember this syntax and compare it with the syntax used in the Java approach to traditional multi-dimensional arrays, which I will discuss in the next module.

The output

Thus, the code in Listing 8 (p. 8) causes the number 5 to be displayed on the computer screen.

Let's recap

To recap, the program named **Array05** creates a two-element array object capable of storing references of type **Object** .

Object is generic

Because **Object** is a completely generic type, each of the elements in the array is capable of storing a reference to any ordinary object, or storing a reference to any array object.

Store reference to ordinary object in generic array

The first element in the array is populated with a reference to an ordinary object instantiated from the class named **Array05** .

(Important: The actual object does not occupy the array element. Rather, the actual object exists someplace else in memory, and a reference to the object occupies the array element.)

Store a reference to an array object in the generic array

The second element in the array of element type **Object** is populated with a reference to another array object capable of containing elements of type **int** .

As above, the actual array object of type **int** does not occupy the second element. Rather, that array object exists someplace else in memory, and a reference to the array object occupies the second element in the array of element type **Object** .

Display some data

After the array object of element type **Object** is created and populated, three print statements are executed to display information about the array object and its contents *(those print statements are shown in Listing 7 (p. 7) and* Listing 8 (p. 8) *)*.

The print statements produce the following output on the computer screen:

```
 Array05@73d6a5
[I@111f71
5
```

Default textual representation of ordinary object

The first line of output is the default textual representation of the ordinary object, achieved by calling the default **toString** method on the reference to the ordinary object.

Default textual representation of array object

The second line of output is the textual representation of the array object of type **int[]** , achieved by calling the default **toString** method on the reference to the array object.

Primitive value stored in array object

The third line of text is the value stored in element index 4 of the **int[]** array object whose reference is stored in element index 1 of the array object of element type **Object** .

Primitive versus non-primitive array element contents

References to objects are stored in the elements of non-primitive array objects. The objects themselves exist somewhere else in memory.

Actual primitive values are stored in the elements of a primitive array object.

Thus, the elements of an array object contain actual primitive values, null references, or actual references to ordinary or array objects, depending on the type of the elements of the array object.

5 Summary

This module begins the discussion of array objects in Java.

The existence of array objects tends to complicate the OOP structure of a Java program otherwise consisting only of ordinary objects.

A completely different syntax is required to create array objects than the syntax normally used to instantiate ordinary objects. Ordinary objects are normally instantiated by applying the **new** operator to the constructor for the target class passing parameters between a pair of matching parentheses.

Array objects *(with default initialization)* are created using the **new** operator, the type of data to be encapsulated in the array, and a square-bracket notation to specify the **length** of the array encapsulated in the object.

Array objects with explicit initialization are created using a comma-separated list of expressions enclosed in curly brackets.

Arrays in Java are objects, which are dynamically created and allocated to dynamic memory.

Like ordinary objects, array objects are accessed via references. The type of such a reference is considered to be **TypeName[]** *(note the empty square brackets in the type specification).*

A reference to an array object can also be assigned to a reference variable of type **Object** *(note the absence of square brackets).* Thus, any of the methods of the **Object** class can be called on a reference to an array object.

As is the case with other languages that support arrays, array objects in Java encapsulate a group of zero or more variables. The variables encapsulated in an array object don't have individual names. Rather, they are accessed using positive integer index values.

The integer indices of a Java array object always extend from **0** to **(n-1)** where **n** is the **length** of the array object.

As of the time of this writing, all array objects in Java encapsulate one-dimensional arrays. However, the component type of an array may itself be an array type. This makes it possible to create array objects whose individual components refer to other array objects. This is the mechanism for creating *multi-dimensional* or *ragged* arrays in Java.

The reference to any array object can be assigned to reference variables of the types **Object** , **Cloneable** , or **Serializable** . If the element type of an array object is one of these types, the elements in the array can refer to:

- Other array objects
- Ordinary objects
- A mixture of the two

6 What's next?

This module has barely scratched the surface in explaining how array objects fit into the grand scheme of things in OOP using Java. In the next module, I will continue the discussion, showing you some of the *(often complex)* aspects of using Java array objects to emulate traditional *multi-dimensional* arrays. I will also show you how to create *ragged* arrays in Java.

7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Array Objects, Part 1
- File: Java1622.htm
- Published: May 15, 2002
- Revised: July 28, 2012

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 9 (p. 11) below.

Listing 9: Complete program listing.

```
/*File Array05.java
Copyright 2002, R.G.Baldwin

This program illustrates storage of
references to ordinary objects and
references to array objects in the
same array object of type Object.

Program output is:

Array05@73d6a5
[I@111f71
5
**********************************/

public class Array05{
  public static void main(
                        String[] args){

    int[] v1 = {1,2,3,4,5};
    Object[] v2 = new Object[2];
    v2[0] = new Array05();
    v2[1] = v1;

    System.out.println(v2[0]);
    System.out.println(v2[1]);
    System.out.println(
                   ((int[])v2[1])[4]);
```

```
  }//end main
}//end class Array05
```

-end-