Java1624: Array Objects, Part 2*

Richard Baldwin

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

Abstract

Baldwin explains the programming details involved in the use of array objects. He shows you three ways to emulate traditional two-dimensional rectangular arrays, and also shows you how to create and use ragged 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. 17)
- What's next? (p. 18)
- Miscellaneous (p. 18)
- Complete program listing (p. 18)

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. 3) . Reference variable declaration..
- Listing 2 (p. 3) . A three-dimensional array object of element type Button.
- Listing 3 (p. 3) . The generic class Object.
- Listing 4 (p. 4) . Primitive type conversions.

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

- Listing 5 (p. 5) . Initialization.
- Listing 6 (p. 5) . Placement of square brackets.
- Listing 7 (p. 5) . Creating the actual array object.
- Listing 8 (p. 6) . An array access expression.
- Listing 9 (p. 7) . Explicit initialization of array elements.
- Listing 10 (p. 7) . Create a two-dimensional rectangular array structure.
- Listing 11 (p. 9) . Using length to populate the leaves of the tree structure.
- Listing 12 (p. 9) . Display leaf object contents.
- Listing 13 (p. 10) . Beginning of a ragged array with two rows and three columns.
- Listing 14 (p. 10) . Create the leaf array objects.
- Listing 15 (p. 11) . Create the array object.
- Listing 16 (p. 12) . Populate the root object.
- Listing 17 (p. 13) . Populate the leaf array objects.
- Listing 18 (p. 13) . Display data in leaf array objects.
- Listing 19 (p. 14) . A triangular array.
- Listing 20 (p. 14) . Populate the leaf array objects.
- Listing 21 (p. 15) . A more general approach.
- Listing 22 (p. 15) . Populate the leaf objects.
- Listing 23 (p. 16) . Beginning of a more general case.
- Listing 24 (p. 16) . Populate the leaf array elements.
- Listing 25 (p. 16) . Display the output.
- Listing 26 (p. 18) . Complete program listing.

3 Preview

This module explains various details regarding the use of array objects in Java, and illustrates many of those details through the use of sample code.

A sample program shows you three ways to emulate traditional two-dimensional rectangular arrays, and also shows you how to create and use ragged arrays.

4 Discussion and sample code

Array objects

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.

The indices of a Java array object

Array objects encapsulate a group of variables, which 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.

Multidimensional arrays

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.

Such a structure of array objects can be thought of as a tree of array objects, with the data being stored in the array objects that make up the leaves of the tree.

Array types

When declaring a reference variable capable of referring to an array object, the array type is declared by writing the name of an element type followed by some number of empty pairs of square brackets []. This is

illustrated in Listing 1 (p. 3) , which declares a reference variable named **v1** , capable of storing a reference to a *two-dimensional* array of type **int** .

Listing 1: Reference variable declaration.

```
int[][] v1;
```

(Note that Listing 1 doesn't really declare a two-dimensional array in the traditional sense of other programming languages. Rather, it declares a reference variable capable of storing a reference to a one-dimensional array object, which in turn is capable of storing references to one-dimensional array objects of type ***int*** *.)*

Multiple pairs of square brackets are allowed

The components in an array object may refer to other array objects. The number of bracket pairs used in the declaration of the reference variable indicates the depth of array nesting *(in the sense that array elements can refer to other array objects).* This is one of the ways that Java implements the concept of traditional multi-dimensional arrays *(I will show you some other ways later in this module).*

The code in Listing 1 shows two levels of nesting for the reference variable of type

```
int[][]
```

Length not part of variable declaration

Note that an array's length is not part of its type or reference variable declaration.

Ragged arrays

Note also that multi-dimensional arrays, when implemented in this fashion, are not required to represent rectangles, cubes, etc. For example, the number of elements in each row of a Java two-dimensional array can be different. Some authors refer to this as a *ragged array* .

Allowable types

The specified element type of an array may be any primitive or reference type. Note, however, that all elements of the array must be of the same type *(consistent with the type-conversion rules discussed below)* .

Listing 2 (p. 3) shows the declaration of a reference variable capable of referring to a three *-dimensional* array object of element type **Button** (***Button*** *is one of the classes in the standard class library).*

Listing 2: A three-dimensional array object of element type Button.

```
Button[][][] v2;
```

Rules of type conversion and assignment compatibility apply

The normal rules of *type conversion* and *assignment compatibility* apply when creating and populating array objects. For example, if the specified type is the name of a non-abstract class, a null reference or a reference to any object instantiated from that class or any subclass of that class may be stored in the array element.

The generic class Object

For example, Listing 3 (p. 3) shows the declaration of a reference variable capable of referring to a one-dimensional array object of element type **Object** .

Since **Object** is the superclass of all other classes, this array object is capable of storing references to objects instantiated from any other class. *(As we saw in the previous module, it is also capable of storing a reference to any other array object as well.)*

Listing 3: The generic class Object.

```
Object[] v3;
```

Primitive type conversions

Similarly, if the declared element type for the array object is one of the primitive types, the elements of the array can be used to store values of any primitive type that is *assignment compatible* with the declared type *(without the requirement for a cast).*

For example, the code in Listing 4 (p. 4) shows the creation of a one-dimensional array object capable of storing values of type **int** . The array object has a length of 3 elements, and the object's reference is stored in a reference variable named **v1** .

Listing 4: Primitive type conversions.

```
int[] v1;
v1 = new int[3];
byte x1 = 127;
short x2 = 16384;
int x3 = 32000;
v1[0] = x1;
v1[1] = x2;
v1[2] = x3;
```

Assignment-compatible assignments

Values of the types **byte** , **short** , and **int** , are stored in the elements of the array object in Listing 4 (p. 4) .

Actual type is lost in the process

It should be noted that the **byte** and **short** values are converted to type **int** as they are stored. When retrieved later, they will be retrieved as type **int** . Any indication that these values were ever of any type other than **int** is lost in the process of storing and retrieving the values.

What about class types?

If the declared element type is the name of a class, *(which may or may not be abstract),* a null reference or a reference to any object instantiated from the class or any subclass of the class may be stored in the array element.

(Obviously you can't store a reference to an object instantiated from an abstract class, because you can't instantiate an abstract class.)

What about an interface type?

If the declared element type is an interface type, a null reference or a reference to any object instantiated from any class that implements the interface can be stored in the array element.

(This is an extremely powerful concept, allowing references to objects instantiated from many different classes to be collected into an array as the interface type.)

Array reference variables

All array objects are accessed via references. A reference variable whose declared type is an array type *does not contain an array.* Rather, it contains either null, or a reference to an array object.

Allocation of memory

Declaring the reference variable does not create an array, nor does it allocate any space for the array components. It simply causes memory to be allocated for the reference variable itself, which may later contain a reference to an array object.

Initialization

In the same sense that it is possible to declare a reference variable for an ordinary object, and initialize it with a reference to an object when it is declared, it is also possible to declare a reference to an array object and initialize it with a reference to an array object when it is declared. This is illustrated in Listing 5 (p. 5) , which shows the following operations combined into a single statement:

- Declaration of a variable to contain a reference to an array object
- Creation of the array object
- Storage of the array object's reference in the reference variable

Listing 5: Initialization.

```
int[] v1 = new int[3];
```

Can refer to different array objects

The **length** of an array is not established when the reference variable is declared. As with references to ordinary objects, a reference to an array object can refer to different array objects at different points in the execution of a program.

For example, a reference variable that is capable of referring to an array of type **int[]** can refer to an array object of a given **length** at one point in the program and can refer to a different array object of the same type but a different **length** later in the program.

Placement of square brackets

When declaring an array reference variable, the square brackets [] may appear as part of the type, or following the variable name, or both. This is illustrated in Listing 6 (p. 5) .

Listing 6: Placement of square brackets.

```
int[][] v1;
int[] v2[];
int v3[][];
```

Type and length

Once an array object is created, its type and length never changes. A reference to a different array object must be assigned to the reference variable to cause the reference variable to refer to an array of different length.

Creating the actual array object

An array object is created by an array creation expression or an array initializer.

An array creation expression *(or an array initializer)* specifies:

- The element type
- The number of levels of nested arrays
- The length of the array for at least one of the levels of nesting

Two valid array creation expressions are illustrated by the statements in Listing 7 (p. 5) .

Listing 7: Creating the actual array object.

```
int[][] v1;
int[] v2[];

v1 = new int[2][3];
v2 = new int[10][];
```

A two-dimensional rectangular array

The third statement in Listing 7 (p. 5) creates an array object of element type **int** with two levels of nesting. This array object can be thought of as a traditional two-dimensional rectangular array having two rows and three columns. *(This is a somewhat arbitrary choice as to which dimension specifies the number of rows and which dimension specifies the number of columns. You may prefer to reverse the two.)*

A ragged array

The fourth statement also creates an array object of element type **int** with two levels of nesting. However, the number of elements in each column is not specified at this point, and it is not appropriate to think of this as a two-dimensional rectangular array. In fact, once the number of elements in each column has been specified, it may not describe a rectangle at all. Some authors refer to an array of this type as a *ragged array.*

The length of the array

The **length** of the array is always available as a final instance variable named **length** . I will show you how to use the value of **length** in a sample program later in this module.

Accessing array elements

An array element is accessed by an *array access expression* . The access expression consists of an expression whose value is an array reference followed by an indexing expression enclosed by matching square brackets.

The expression in parentheses in Listing 8 (p. 6) illustrates an array access expression *(or perhaps two concatenated array access expressions).*

Listing 8: An array access expression.

```
int[][] v1 = new int[2][3];
System.out.println(v1[0][1]);
```

First-level access

This *array access expression* first accesses the contents of the element at index 0 in the array object referred to by the reference variable named **v1** . This element contains a reference to a second array object *(note the double matching square brackets, [][] in the declaration of the variable named* ***v1*** *).*

Second-level access

The *array access expression* in Listing 8 (p. 6) uses that reference to access the value stored in the element at index value 1 in the second array object. That value is then passed to the **println** method for display on the standard output device.

(In this case, the value 0 is displayed, because array elements are automatically initialized to default values when the array object is created. The default value for all primitive numeric values is zero.)

Zero-based indexing

All array indexes in Java begin with **0** . An array with length **n** can be indexed by the integers **0** to **(n-1)** . Array accesses are checked at runtime. If an attempt is made to access the array with any other index value, an **ArrayIndexOutOfBoundsException** will be thrown.

Index value types

Arrays must be indexed by integer values of the following types: **int** , **short** , **byte** , or **char** . For any of these types other than **int** , the value will be promoted to an **int** and used as the index.

An array cannot be accessed using an index of type **long** . Attempting to do so results in a compiler error.

Default initialization

If the elements in an array are not purposely initialized when the array is created, the array elements will be automatically initialized with default values. The default values are:

- All reference types: null

- Primitive numeric types: 0
- Primitive boolean type: false
- Primitive char type: the Unicode character with 16 zero-valued bits

Explicit initialization of array elements

The values in the array elements may be purposely initialized when the array object is created using a comma-separated list of expressions enclosed by matching curly brackets. This is illustrated in Listing 9 (p. 7) .

Listing 9: Explicit initialization of array elements.

```
int[] v1 = {1,2,3,4,5};
```

No new operator

Note that this format does not use the **new** operator. Also note that the expressions in the list may be much more complex than the simple literal values shown in Listing 9 (p. 7) .

Length and order

When this format is used, the length of the constructed array will equal the number of expressions in the list.

The expressions in an array initializer are executed from left to right in the order that they occur in the source code. The first expression specifies the value at index value zero, and the last expression specifies the value at index value n-1 *(where n is the length of the array).*

Each expression must be assignment-compatible with the array's component type, or a compiler error will occur.

A sample program

The previous paragraphs in this module have explained some of the rules and characteristics regarding array objects. They have also illustrated some of the syntax involved in the use of array objects in Java.

More powerful and complex

Many aspiring Java programmers find the use of array objects to be something less than straightforward, and that is understandable. In fact, Java array objects are somewhat more powerful than array structures in many other programming languages, and this power often manifests itself in additional complexity.

A traditional two-dimensional rectangular array

Some of that complexity is illustrated by the program named **Array07** , shown in Listing 26 (p. 18) near the end of this module. This program illustrates three different ways to accomplish essentially the same task using array objects in Java. That task is to emulate a traditional two-dimensional rectangular array as found in other programming languages. Two of the ways that are illustrated are essentially ragged arrays with sub-arrays having equal length.

Ragged arrays

The program also illustrates two different ways to work with array objects and ragged arrays.

Will discuss in fragments

As is my practice, I will discuss and explain the program in fragments.

All of the interesting code in this program is contained in the **main** method, so I will begin my discussion with the first statement in the **main** method.

Create a two-dimensional rectangular array structure

Listing 10 (p. 7) creates an array structure that emulates a traditional rectangular array with two rows and three columns.

Listing 10: Create a two-dimensional rectangular array structure.

```
Object[][] v1 = new Object[2][3];
```

(Note that unlike the ragged array structures to be discussed later, this approach requires all rows to be the same length and all columns to be the same length.)

Reference variable declaration

The code to the left of the equal sign *(=)* in Listing 10 (p. 7) declares a reference variable named **v1** . This reference variable is capable of holding a reference to an array object whose elements are of the type **Object[]** ..

In other words, this reference variable is capable of

- holding a reference to an array object,
- whose elements are capable of holding references to other array objects,
- whose elements are of type **Object** .

Two levels of nesting

The existence of double matching square brackets in the variable declaration in Listing 10 (p. 7) indicates two levels of nesting.

Restrictions

The elements in the array object referred to by **v1** can only hold references to other array objects whose element type is **Object** *(or references to array objects whose element type is a subclass of* ***Object*** *).*

The elements in the array object referred to by **v1** cannot hold references to ordinary objects instantiated from classes, or array objects whose element type is a primitive type.

In other words, the elements in the array object referred to by **v1** can only hold references to other array objects. The element types of those array objects must be *assignment compatible* with the type **Object** *(this includes interface types and class types but not primitive types).*

A tree of empty array objects

The code to the right of the equal sign *(=)* in Listing 10 (p. 7) creates a *tree structure* of array objects. The object at the root of the tree is an array object of type **Object[]** , having two elements *(a* ***length*** *of two).*

The reference variable named **v1** refers to the array object that forms the root of the tree.

Each of the two elements in the root array object is initialized with a reference to another array object.

(These two objects might be viewed as sub-arrays, or as child nodes in the tree structure).

Each of the child nodes is an array object of type **Object** „ and has a **length** of three.

Each element in each of the two child node array objects is initialized to the value **null** *(this is the default initialization for array elements of reference types that don't yet refer to an object).*

Recap

To recap, the reference variable named **v1** contains a reference to a two-element, one-dimensional array object. Each element in that array object is capable of storing a reference of type **Object[]** *(a reference to another one-dimensional array object of type* ***Object*** *).*

Two sub-array objects

Two such one-dimensional sub-array *(or child node)* objects, of element type **Object** , are created. References to the two sub-array objects are stored in the elements of the two-element array object at the root of the tree.

Each of the sub-array objects has three elements. Each element is capable of storing a reference to an object as type **Object** .

The leaves of the tree

These two sub-array objects might be viewed as the leaves of the tree structure.

Initialize elements to null

However, the objects of type **Object** don't exist yet. Therefore, each element in each of the sub-array objects is automatically initialized to **null** .

Arrays versus sub-arrays

Note that there is no essential difference between an array object and a sub-array object in the above discussion. The use of the sub prefix is used to indicate that an ordinary array object belongs to another array object, because the reference to the sub-array object is stored in an element of the owner object.

Many dimensions are possible

Multi-dimensional arrays of any *(reasonable)* depth can be emulated in this manner. An array object may contain references to other array objects, which may contain references to other array objects, and so on.

The leaves of the tree structure

Eventually, however, the elements of the leaves in the tree structure must be specified to contain either primitive values or references to ordinary objects. This is where the data is actually stored.

(Note however, that if the leaves are specified to contain references of type ***Object*** *, they may contain references to other array objects of any type, and the actual data could be stored in those array objects.)*

The length of an array

Every array object contains a public final instance variable named **length** , which contains an integer value specifying the number of elements in the array.

Once created, the length of the array encapsulated in an array object cannot change. Therefore, the value of **length** specifies the length of the array throughout the lifetime of the array object.

Using length to populate the leaves of the tree structure

The value of **length** is very handy when processing array objects. This is illustrated in Listing 11 (p. 9) , which uses a nested **for** loop to populate the elements in the leaves of the tree structure referred to by **v1** . *(The elements in the leaf objects are populated with references to objects of type* ***Integer*** *.)*

Listing 11: Using length to populate the leaves of the tree structure.

```
for(int i=0;i<v1.length;i++){
  for(int j=0;j<v1[i].length;j++){
    v1[i][j] =
          new Integer((i+1)*(j+1));
  }//end inner loop
}//end outer loop
```

Using length in loop's conditional expressions

Hopefully by now you can read and understand this code without a lot of help from me. I will point out, however, that the value returned by **v1.length** *(in the conditional expression for the outer loop)* is the number of leaves in the tree structure *(this tree structure has two leaves).*

I will also point out that the value returned by **v1[i].length** *(in the conditional expression for the inner loop)* is the number of elements in each leaf array object *(each leaf object in this tree structure has three elements).*

Finally, I will point out that the expression **v1[i][j]** accesses the ***jth*** element in the ***ith*** leaf, or sub-array. In the traditional sense of a rectangular array, this could be thought of as the ***jth*** column of the ***ith*** row. This mechanism is used to store object references in each element of each of the leaf array objects.

Populate with references to Integer objects

Thus, each element in each leaf array object is populated with a reference to an object of the type **Integer** . Each object of the type **Integer** encapsulates an **int** value calculated from the current values of the two loop counters.

Display leaf object contents

In a similar manner, the code in Listing 12 (p. 9) uses the **length** values in the conditional expressions of nested **for** loops to access the references stored in the elements of the leaf array objects, and to use those references to access and display the values encapsulated in the **Integer** objects whose references are stored in those elements.

Listing 12: Display leaf object contents.

```
      for(int i=0;i<v1.length;i++){
    for(int j=0;j<v1[i].length;j++){
      System.out.print(
                     v1[i][j] + " ");
    }//end inner loop
    System.out.println();//new line
  }//end outer loop
```

The rectangular output

The code in Listing 12 (p. 9) produces the following output on the screen.

```
  1 2 3
2 4 6
```

As you can see, this emulates a traditional two-dimensional array having two rows and three columns.

A ragged array with two rows and three columns

The second approach to emulating a traditional two-dimensional rectangular array will create a *ragged array* where each row is the same length.

(It is very important to note that, unlike this case, with a ragged array, the number of elements in each row or the number of elements in each column can be different.)

The most significant thing about this approach is the manner in which the tree of array objects is created *(see Listing 13 (p. 10)).*

Listing 13: Beginning of a ragged array with two rows and three columns.

```
Object[][] v2 = new Object[2][];
```

Three statements required

With this approach, three statements are required to replace one statement from the previous approach. *(Two additional statements are shown in Listing 14 (p. 10) .)*

A single statement in the previous approach *(Listing 10 (p. 7))* created all three array objects required to construct the tree of array objects, and initialized the elements in the leaf array objects with **null** values.

Create only the root array object

However, the boldface code in Listing 13 (p. 10) creates only the array object at the root of the tree. That array object is an array object having two elements capable of storing references of type **Object[]** .

Empty square brackets

If you compare this statement with the statement in Listing 10 (p. 7) , you will notice that the right-most pair of square brackets in Listing 13 (p. 10) is empty. Thus, Listing 13 (p. 10) creates only the array object at the root of the tree, and initializes the elements in that array object with **null** values.

Leaf array objects don't exist yet

The leaf array objects don't exist at the completion of execution of the statement in Listing 13 (p. 10) .

Create the leaf array objects

The statements in Listing 14 (p. 10) create two array objects of element type **Object** .

Listing 14: Create the leaf array objects.

```
v2[0] = new Object[3];
v2[1] = new Object[3];
```

Save the references to the leaves

References to these two leaf objects are stored in the elements of the array object at the root of the tree, *(which was created in* Listing 13 (p. 10) *)*. Thus, these two array objects become the leaves of the tree structure of array objects.

This completes the construction of the tree structure. Each element in each leaf object is initialized with **null** .

Why bother?

You might ask why I would bother to use this approach, which requires three statements in place of only one statement in the previous approach.

The answer is that I wouldn't normally use this approach if my objective were to emulate a traditional rectangular array. However, this approach is somewhat more powerful than the previous approach.

The lengths of the leaf objects can differ

With this approach, the **length** values of the two leaf array objects need not be the same. Although I caused the **length** value of the leaf objects to be the same in this case, I could just as easily have caused them to be different lengths *(I will illustrate this capability later in the program).*

Populate and display the data

If you examine the complete program in Listing 26 (p. 18) near the end of the module, you will see that nested **for** loops, along with the value of **length** was used to populate and display the contents of the leaf array objects. Since that portion of the code is the same as with the previous approach, I won't show and discuss it here.

The rectangular output

This approach produced the following output on the screen, *(which is the same as before):*

```
 1 2 3
2 4 6
```

Now for something really different

The next approach that I am going to show you for emulating a two-dimensional rectangular array is significantly different from either of the previous two approaches.

Not element type Object[]

In this approach, I will create a one-dimensional array object of element type **Object** *(not element type* ***Object[]*** *)* . I will populate the elements of that array object with references to other array objects of element type **Object** . In doing so, I will create a tree structure similar to those discussed above.

The length of the leaf objects

As with the second approach above, the array objects that make up the leaves of the tree can be any length, but I will make them the same length in order to emulate a traditional rectangular two-dimensional array.

Create the array object

First consider the statement shown in Listing 15 (p. 11) . Compare this statement with the statements shown earlier in Listing 10 (p. 7) and Listing 13 (p. 10) .

Listing 15: Create the array object.

```
Object[] v3 = new Object[2];
```

No double square brackets

Note in particular that the statement in Listing 15 (p. 11) does not make use of double square brackets, as was the case in Listing 10 (p. 7) and Listing 13 (p. 10) . Thus, the statement show in Listing 15 (p. 11) is entirely different from the statements shown in Listing 10 (p. 7) and Listing 13 (p. 10) .

Declare a reference variable

That portion of the statement to the left of the equal sign *(=)* declares a reference variable capable of storing a reference to an array object whose elements are capable of storing references of the type **Object** *(not type* ***Object[]*** *as in the previous examples).*

Refer to the root object

This reference variable will refer to an array object that forms the root of the tree structure. However, the root object in this case will be considerably different from the root objects in the previous two cases.

In the previous two cases, the elements of the root object were required to store references of type **Object[]** *(note the square brackets).* In other words, an array object whose elements are of type **Object[]** can only store references to other array objects whose elements are of type **Object** .

A more general approach

However, an array object whose elements are of type **Object** *(as is the case here),* can store:

- References to any object instantiated from any class
- References to array objects whose elements are of any type *(primitive or reference)*
- A mixture of the two kinds of references .

Therefore, this is a much more general, and much more powerful approach.

A price to pay

However, there is a price to pay for the increased generality and power. In particular, the programmer who uses this approach must have a much better understanding of Java object-oriented programming concepts than the programmer who uses the two approaches discussed earlier in this module.

Particularly true relative to first approach

This is particularly true relative to the first approach discussed earlier. That approach is sufficiently similar to the use of multi-dimensional arrays in other languages that a programmer with little understanding of Java object-oriented programming concepts can probably muddle through the syntax based on prior knowledge. However, it is unlikely that a programmer could muddle through this approach without really understanding what she is doing.

Won't illustrate true power

Although this approach is very general and very powerful, this sample program won't attempt to illustrate that power and generality. Rather, this sample program will use this approach to emulate a traditional two-dimensional rectangular array just like the first two approaches discussed earlier. *(Later, I will also use this approach for a couple of ragged arrays.)*

Populate the root object

The two statements in Listing 16 (p. 12) create two array objects, each having three elements. Each element is capable of storing a reference to any object that is *assignment compatible* with the **Object** type.

(Assignment compatibility includes a reference to any object instantiated from any class, or a reference to any array object of any type (including primitive types), or a mixture of the two.)

Listing 16: Populate the root object.

```
v3[0] = new Object[3];
v3[1] = new Object[3];
```

References to the two new array objects are stored in the elements of the array object that forms the root of the tree structure. The two new array objects form the leaves of the tree structure.

Populate the leaf array objects

As in the previous two cases, the code in Listing 17 (p. 13) uses nested **for** loops to populate the array elements in the leaf objects with references to new **Integer** objects. *(The* ***Integer*** *objects encapsulate* ***int*** *values based on the loop counter values for the outer and inner loops.)*

Listing 17: Populate the leaf array objects.

```
for(int i=0;i<v3.length;i++){
  for(int j=0;j<((Object[])v3[i]).length;j++){
    ((Object[])v3[i])[j] = new Integer((i+1)*(j+1));
  }//end inner loop
}//end outer loop
```

Added complexity

The added complexity of this approach manifests itself in

- The *cast operators* shown in boldface Italics in Listing 17 (p. 13)
- The attendant required grouping of terms within parentheses

Inside and outside the parentheses

Note that within the inner loop, one of the square-bracket accessor expressions is inside the parentheses and the other is outside the parentheses.

Why are the casts necessary?

The casts are necessary to convert the references retrieved from the array elements from type **Object** to type **Object[]** . For example, the reference stored in **v3[i]** is stored as type **Object** .

Get length of leaf array object

The cast in the following expression converts that reference to type **Object[]** before attempting to get the value of **length** belonging to the array object whose reference is stored there.

```
((Object[])v3[i]).length
```

Assign a value to an element in the leaf array object

Similarly, the following expression converts the reference stored in **v3[i]** from type **Object** to type **Object[]** . Having made the conversion, it then accesses the **jth** element of the array object whose reference is stored there *(in order to assign a value to that element).*

```
((Object[])v3[i])[j]=
```

Display data in leaf array objects

Listing 18 (p. 13) uses similar casts to get and display the values encapsulated in the **Integer** objects whose references are stored in the elements of the leaf array objects.

Listing 18: Display data in leaf array objects.

```
    for(int i=0;i<v3.length;i++){
  for(int j=0;j<((Object[])v3[i]).length;j++){
    System.out.print(((Object[])v3[i])[j] + " ");
  }//end inner loop
  System.out.println();//new line
}//end outer loop
```

The rectangular output

This approach produced the following output on the screen, *(which is the same as the previous two approaches):*

```
  1 2 3
2 4 6
```

Ragged arrays

All the code in the previous three cases has been used to emulate a traditional rectangular two-dimensional array. In the first case, each row was required to have the same number of elements by the syntax used to create the tree of array objects.

In the second and third cases, each row was not required to have the same number of elements, but they were programmed to have the same number of elements in order to emulate a rectangular two-dimensional array.

A triangular array, sort of ...

Now I am going to show you some cases that take advantage of the *ragged-array* capability of Java array objects. In the next case, *(beginning with Listing 19 (p. 14))*, I will create a ragged array having two rows. The first row will have two elements and the second row will have three elements. *(This array object might be thought of as being sort of triangular.)*

Listing 19: A triangular array.

```
Object[][] v4 = new Object[2][];
v4[0] = new Object[2];
v4[1] = new Object[3];
```

You have seen this before

You saw code like this in the second case discussed earlier. However, in that case, the second and third statements created new array objects having the same length. In this case, the second and third statements create array objects having different lengths. This is one of the ways to create a ragged array in Java *(you will see another way in the next case that I will discuss)*.

Populate the leaf array objects

Listing 20 (p. 14) populates the elements of the leaf array objects with references to objects of the class **Integer** .

Listing 20: Populate the leaf array objects.

```
    for(int i=0;i<v4.length;i++){
  for(int j=0;j<v4[i].length;j++){
    v4[i][j] =
          new Integer((i+1)*(j+1));
  }//end inner loop
}//end outer loop
```

You have seen this before also

You have also seen the code in Listing 20 (p. 14) before. I repeated it here because this case clearly emphasizes the value of the **length** constant that is available in all Java array objects. In the earlier case, the **length** of the two leaf array objects was the same, so it would have been feasible to simply hard-code that value into the conditional expression of the inner **for** loop.

The length is not the same now

However, in this case, the **length** of the two leaf array objects is not the same. Therefore, it wouldn't work to hard-code a limit into the conditional expression of the inner **for** loop. However, because the **length** of each leaf array object is available as a public member of the array object, that value can be used to control the number of iterations of the inner loop for each separate leaf array object.

The triangular output

The next section of code in the program shown in Listing 26 (p. 18) near the end of the module uses the same code as before to display the **int** values encapsulated in the **Integer** objects whose references are stored in the leaf array objects. Since it is the same code as before, I won't repeat it here.

The output produced by this case is shown below:

```
 1 2
2 4 6
```

Note that this is not the same as before, and this output does not describe a rectangular array. Rather, it describes a ragged array where the rows are of different lengths.

(As I indicated earlier, it is sort of triangular. However, it could be any shape that you might want it to be.)

A more general approach

The next case, shown in Listing 21 (p. 15) , is the same as the third case discussed earlier, except that the lengths of the leaf array objects are not the same.

As before, this case creates a one-dimensional array object of type **Object** *(having two elements)* that forms the root of a tree. Each element in the root object contains a reference to another array object of type **Object** .

One of those leaf objects has two elements and the other has three elements, thus producing a ragged array *(you could make the lengths of those objects anything that you want them to be).*

Listing 21: A more general approach.

```
Object[] v5 = new Object[2];
v5[0] = new Object[2];
v5[1] = new Object[3];
```

Populate the leaf objects

As before, the elements in the leaf array objects are populated with references to objects of the class **Integer** , which encapsulate **int** values based on the current value of the loop counters. This is shown in Listing 22 (p. 15) .

Listing 22: Populate the leaf objects.

```
    for(int i=0;i<v5.length;i++){
  for(int j=0;
        j<((Object[])v5[i]).length;
                                j++){
    ((Object[])v5[i])[j] =
          new Integer((i+1)*(j+1));
  }//end inner loop
}//end outer loop
```

Same code as before

This is the same code that you saw in Listing 17 (p. 13) . I repeated it here to emphasize the requirement for casting .

Display the data

This case uses the same code as Listing 18 (p. 13) to display the **int** values encapsulated by the **Integer** objects whose references are stored in the elements of the leaf array objects. I won't repeat that code here.

The triangular output

The output produced by this case is shown below:

```
 1 2
2 4 6
```

Note that this is the same as the case immediately prior to this one. Again, it does not describe a rectangular array. Rather, it describes a ragged array where the rows are of different lengths.

A more general case

I'm going to show you one more general case for a ragged array. This case illustrates a more general approach. In this case, I will create a one-dimensional array object of element type **Object** . I will populate the elements of that array object with references to other array objects. These array objects will be the leaves of the tree.

Leaf array objects are type int

In this case, the leaves won't be of element type **Object** . Rather, the elements in the leaf objects will be designed to store primitive **int** values.

(An even more general case would be to populate the elements of the root object with references to a mixture of objects of class types, interface types, and array objects where the elements of the array objects are designed to store primitives of different types, and references of different types. Note, however, each leaf array object must be designed to store a single type, but will accept for storage any type that is assignment-compatible with the specified type for the array object.)

This case begins in Listing 23 (p. 16) , which creates the root array object, and populates its elements with references to leaf array objects of type **int** .

Listing 23: Beginning of a more general case .

```
Object[] v6 = new Object[2];
v6[0] = new int[7];
v6[1] = new int[3];
```

Leaf objects are different lengths

One of the leaf array objects has a length of 7. The other has a length of 3.

Populate the leaf array elements

Listing 24 (p. 16) populates the elements in the leaf array objects with values of type **int** .

Listing 24: Populate the leaf array elements.

```
    for(int i=0;i<v6.length;i++){
  for(int j=0;j<((int[])v6[i]).length;j++){
    ((int[])v6[i])[j] = (i+2)*(j+2);
  }//end inner loop
}//end outer loop
```

Similar to previous code

The code in Listing 24 (p. 16) is similar to code that you saw earlier. The differences are:

- Cast is to type **int[]** instead of **object[]**
- Values assigned are type **int** instead of references to **Integer** objects

Display the output

Finally, Listing 25 (p. 16) displays the **int** values stored in the elements of the leaf array objects.

Listing 25: Display the output.

```
    for(int i=0;i<v6.length;i++){
  for(int j=0;j<((int[])v6[i]).length;j++){
    System.out.print(((int[]
```

The code in Listing 25 (p. 16) is very similar to what you have seen before, and there should be no requirement for an explanation of this code.

The code in Listing 25 (p. 16) produces the following output:

```
    4 6 8 10 12 14 16
6 9 12
```

I will leave it as an exercise for the student to correlate the output with the code.

5 Summary

When declaring a reference variable capable of referring to an array object, the array type is declared by writing the name of an element type followed by some number of empty pairs of square brackets [].

The components in an array object may refer to other array objects. The number of bracket pairs used in the declaration of the reference variable indicates the depth of array nesting *(in the sense that array elements can refer to other array objects).*

An array's length is not part of its type or reference variable declaration.

Multi-dimensional arrays are not required to represent rectangles, cubes, etc. They can be *ragged.*

The normal rules of *type conversion* and *assignment compatibility* apply when creating and populating array objects.

Object is the superclass of all other classes. Therefore, an array of element type **Object** is capable of storing references to objects instantiated from any other class. The type declaration for such an array object would be **Object[]** .

An array of element type **Object** is also capable of storing references to any other array object.

If the declared element type for the array object is one of the primitive types, the elements of the array can be used to store values of any primitive type that is *assignment compatible* with the declared type *(without the requirement for a cast).*

If the declared element type is the name of a class, *(which may or may not be abstract),* a null reference or a reference to any object instantiated from the class or any subclass of the class may be stored in the array element.

If the declared element type is an interface type, a null reference or a reference to any object instantiated from any class that implements the interface can be stored in the array element.

A reference variable whose declared type is an array type *does not contain an array.* Rather, it contains either null, or a reference to an array object. Declaring the reference variable does not create an array, nor does it allocate any space for the array components.

It is possible to declare a reference to an array object and initialize it with a reference to an array object when it is declared.

A reference to an array object can refer to different array objects *(of the same element type and different lengths)* at different points in the execution of a program.

When declaring an array reference variable, the square brackets [] may appear as part of the type, or following the variable name, or both.

Once an array object is created, its type and length never changes.

An array object is created by an array creation expression or an array initializer.

An array creation expression *(or an array initializer)* specifies:

- The element type
- The number of levels of nested arrays
- The length of the array for at least one of the levels of nesting

The length of the array is always available as a final instance variable named **length** .

An array element is accessed by an expression whose value is an array reference followed by an indexing expression enclosed by matching square brackets.

If an attempt is made to access the array with an invalid index value, an **ArrayIndexOutOfBoundsException** will be thrown.

Arrays must be indexed by integer values of the types **int** , **short** , **byte** , or **char** . An array cannot be accessed using an index of type **long** .

If the elements in an array are not purposely initialized when the array is created, the array elements will be automatically initialized with default values.

The values in the array elements may be purposely initialized when the array object is created using a comma-separated list of expressions enclosed by matching curly brackets.

The program in this module illustrated three different ways to emulate traditional rectangular two-dimensional arrays.

The program also illustrated two different ways to create and work with ragged arrays.

6 What's next?

In the next module, I will provide so additional information about array objects, and then illustrate the use of the classes named **Array** and **Arrays** for the creation and manipulation of array objects.

7 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Array Objects, Part 2
- File: Java1624.htm
- Published: May 22, 2002
- Revised: July 29, 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 26 (p. 18) below.

Listing 26: Complete program listing.

```
    /*File Array07.java
Copyright 2002, R.G.Baldwin

This program illustrates three
different ways to emulate a traditional
rectangular array in Java.  Two of
those ways are essentially ragged
arrays with equal-length sub arrays.

The program also illustrates two ways
to create ragged arrays in Java.

Tested using JDK 1.3 under Win 2000.
**************************************/

public class Array07{
  public static void main(
                        String[] args){

    //Create an array structure that
    // emulates a traditional
    // rectangular array with two rows
    // and three columns.  This
    // approach requires all rows to
    // be the same length.
    Object[][] v1 = new Object[2][3];
    //Populate the array elements with
    // references to objects of type
    // Integer.
    for(int i=0;i<v1.length;i++){
      for(int j=0;j<v1[i].length;j++){
        v1[i][j] =
              new Integer((i+1)*(j+1));
      }//end inner loop
    }//end outer loop
    //Display the array elements
    for(int i=0;i<v1.length;i++){
      for(int j=0;j<v1[i].length;j++){
        System.out.print(
                        v1[i][j] + " ");
      }//end inner loop
      System.out.println();//new line
    }//end outer loop
    System.out.println();//new line

    //Create a ragged array with two
    // rows.  The first row has three
    // columns and the second row has
    // three columns.  The length of
    // each row could be anything, but
    // was set to three to match the
```

```
// above array structure.
Object[][] v2 = new Object[2][];
v2[0] = new Object[3];
v2[1] = new Object[3];
//Populate the array elements with
// references to objects of type
// Integer.
for(int i=0;i<v2.length;i++){
  for(int j=0;j<v2[i].length;j++){
    v2[i][j] =
          new Integer((i+1)*(j+1));
  }//end inner loop
}//end outer loop
//Display the array elements
for(int i=0;i<v2.length;i++){
  for(int j=0;j<v2[i].length;j++){
    System.out.print(
                    v2[i][j] + " ");
  }//end inner loop
  System.out.println();//new line
}//end outer loop
System.out.println();//new line

//Create a one-dimensional array
// of type Object, which contains
// references to array objects of
// type Object.  The secondary
// array objects could be of any
// length, but were set to three
// to match the above array
// structure.
Object[] v3 = new Object[2];
v3[0] = new Object[3];
v3[1] = new Object[3];
//Populate the array elements with
// references to objects of type
// Integer.
for(int i=0;i<v3.length;i++){
  for(int j=0;
        j<((Object[])v3[i]).length;
                              j++){
    ((Object[])v3[i])[j] =
          new Integer((i+1)*(j+1));
  }//end inner loop
}//end outer loop
//Display the array elements
for(int i=0;i<v3.length;i++){
  for(int j=0;
        j<((Object[])v3[i]).length;
                              j++){
    System.out.print(
```

```
        ((Object[])v3[i])[j] + " ");
    }//end inner loop
    System.out.println();//new line
  }//end outer loop
  System.out.println();//new line

  //Create a ragged array with two
  // rows.  The first row has two
  // columns and the second row has
  // three columns.
  Object[][] v4 = new Object[2][];
  v4[0] = new Object[2];
  v4[1] = new Object[3];
  //Populate the array elements with
  // references to objects of type
  // Integer.
  for(int i=0;i<v4.length;i++){
    for(int j=0;j<v4[i].length;j++){
      v4[i][j] =
            new Integer((i+1)*(j+1));
    }//end inner loop
  }//end outer loop
  //Display the array elements
  for(int i=0;i<v4.length;i++){
    for(int j=0;j<v4[i].length;j++){
      System.out.print(
                      v4[i][j] + " ");
    }//end inner loop
    System.out.println();//new line
  }//end outer loop
  System.out.println();//new line

  //Create a one-dimensional array
  // of type Object, which contains
  // references to array objects of
  // type Object.  The secondary
  // array objects could be of any
  // length, but were set to two and
  // three to match the ragged array
  // above.
  Object[] v5 = new Object[2];
  v5[0] = new Object[2];
  v5[1] = new Object[3];
  //Populate the array elements with
  // references to objects of type
  // Integer.
  for(int i=0;i<v5.length;i++){
    for(int j=0;
          j<((Object[])v5[i]).length;
                                  j++){
      ((Object[])v5[i])[j] =
```

```
              new Integer((i+1)*(j+1));
      }//end inner loop
    }//end outer loop
    //Display the array elements
    for(int i=0;i<v5.length;i++){
      for(int j=0;
            j<((Object[])v5[i]).length;
                                  j++){
        System.out.print(
           ((Object[])v5[i])[j] + " ");
      }//end inner loop
      System.out.println();//new line
    }//end outer loop
    System.out.println();

    //Create a one-dimensional array
    // of type int, which contains
    // references to array objects of
    // type Object.  The secondary
    // array objects could be of any
    // length.
    Object[] v6 = new Object[2];
    v6[0] = new int[7];
    v6[1] = new int[3];
    //Now illustrate that the elements
    // of the leaves of a ragged array
    // implemented in this manner can
    // contain primitive values.
    // Populate the array elements with
    // type int.
    for(int i=0;i<v6.length;i++){
      for(int j=0;
            j<((int[])v6[i]).length;
                                  j++){
        ((int[])v6[i])[j] = (i+2)*(j+2);
      }//end inner loop
    }//end outer loop
    //Display the array elements
    for(int i=0;i<v6.length;i++){
      for(int j=0;
            j<((int[])v6[i]).length;
                                  j++){
        System.out.print(
           ((int[])v6[i])[j] + " ");
      }//end inner loop
      System.out.println();//new line
    }//end outer loop

  }//end main
}//end class Array07
```

-end-