JB0240: Java OOP: Arrays and Strings*

Richard Baldwin

Abstract

This module takes a preliminary look at arrays and strings. More in-depth discussions will be provided in future modules.

1 Table of Contents

2 Preface

This module takes a preliminary look at arrays and strings. More in-depth discussions will be provided in future modules. For example, you will find a more in-depth discussions of array objects in the following modules:

- Java OOP: Array Objects, Part 1 [1]
- Java OOP: Array Objects, Part 2 [2]
- Java OOP: Array Objects, Part 3 [3]

*Version 1.3: Nov 25, 2012 10:45 pm -0600
†http://creativecommons.org/licenses/by/3.0/
[1]http://cnx.org/content/m44198
[2]http://cnx.org/content/m44199
[3]http://cnx.org/content/m44200

2.1 Viewing tip

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

2.1.1 Figures

2.1.2 Listings

3 Introduction

The first step

The first step in learning to use a new programming language is usually to learn the foundation concepts such as variables, types, expressions, flow-of-control, arrays, strings, etc. This module concentrates on arrays and strings.

Array and String types

Java provides a type for both arrays and strings from which objects of the specific type can be instantiated. Once instantiated, the methods belonging to those types can be called by way of the object.

4 Arrays

Arrays and Strings

Java has a true array type and a true **String** type with protective features to prevent your program from writing outside the memory bounds of the array object or the **String** object. Arrays and strings are true objects.

Declaring an array

You must declare an array before you can use it. *(More properly, you must declare a reference variable to hold a reference to the array object.)* In declaring the array, you must provide two important pieces of information:

- the name of a variable to hold a reference to the array object
- the type of data to be stored in the elements of the array object

Different declaration formats

A reference variable capable of holding a reference to an array object can be declared using either format shown in Figure 1 (p. 3) .

Formats for declaring a reference variable for an array object.

```
int[] myArray;
int myArray[];
```

Figure 1: Formats for declaring a reference variable for an array object.

Declaration does not allocate memory

As with other objects, the declaration of the reference variable does not allocate memory to contain the array data. Rather it simply allocates memory to contain a reference to the array.

Allocating memory for the array object

Memory to contain the array object must be allocated from dynamic memory using statements such as those shown in Figure 2 (p. 3) .

Allocating memory for the array object.

```
int[] myArrayX = new int[15];
int myArrayY[] = new int[25];

int[] myArrayZ = {3,4,5};
```

Figure 2: Allocating memory for the array object.

The statements in Figure 2 (p. 3) simultaneously declare the reference variable and cause memory to be allocated to contain the array.

Also note that the last statement in Figure 2 (p. 3) is different from the first two statements. This syntax not only sets aside the memory for the array object, the elements in the array are initialized by evaluating the expressions shown in the coma-separated list inside the curly brackets.

On the other hand, the array elements in the first two statements in Figure 2 (p. 3) are automatically initialized with the default value for the type.

Declaration and allocation can be separated

It is not necessary to combine these two processes. You can execute one statement to declare the reference

variable and another statement to cause the array object to be instantiated some time later in the program as shown in Figure 3 (p. 4) .

Declaration and instantiation can be separated.

```
int[] myArray;
. . .
myArray = new int[25];
```

Figure 3: Declaration and instantiation can be separated.

Causing memory to be set aside to contain the array object is commonly referred to as instantiating the array object *(creating an instance of the array object)* .

If you prefer to declare the reference variable and instantiate the array object at different points in your program, you can use the syntax shown in Figure 3 (p. 4) . This pattern is very similar to the declaration and instantiation of all objects.

General syntax for combining declaration and instantiation

The general syntax for declaring and instantiating an array object is shown in Figure 4 (p. 4) .

General syntax for combining declaration and instantiation.

```
typeOfElements[] nameOfRefVariable =
          new typeOfElements[sizeOfArray]
```

Figure 4: General syntax for combining declaration and instantiation.

Accessing array elements

Having instantiated an array object, you can access the elements of the array using indexing syntax that is similar to many other programming languages. An example is shown in Figure 5 (p. 5) .

An example of array indexing syntax.

```
myArray[5] = 6;

myVar = myArray[5];
```

Figure 5: An example of array indexing syntax.

The value of the first index

Array indices always begin with 0.

The length property of an array

The code fragment in Figure 6 (p. 5) illustrates another interesting aspect of arrays. *(Note the use of* ***length*** *in the conditional clause of the* ***for*** *loop.)*

The use of the length property in the conditional clause of a for loop.

```
for(int cnt = 0; cnt < myArray.length; cnt++)
  myArray[cnt] = cnt;
```

Figure 6: The use of the length property in the conditional clause of a for loop.

All array objects have a **length** property that can be accessed to determine the number of elements in the array. *(The number of elements cannot change once the array object is instantiated.)*

Types of data that you can store in an array object

Array elements can contain any Java data type including primitive values and references to ordinary objects or other array objects.

Constructing multi-dimensional arrays

All array objects contains a one-dimensional array structure. You can create multi-dimensional arrays by causing the elements in one array object to contain references to other array objects. In effect, you can create a tree structure of array objects that behaves like a multi-dimensional array.

Odd-shaped multi-dimensional arrays

The program **array01** shown in Listing 1 (p. 6) illustrates an interesting aspect of the Java arrays. Java can produce multi-dimensional arrays that can be thought of as an array of arrays. However, the secondary arrays need not all be of the same size.

In the program shown in Listing 1 (p. 6) , a two-dimensional array of integers is declared and instantiated with the primary size *(size of the first dimension)* being three. The sizes of the secondary dimensions *(sizes of each of the sub-arrays)* is 2, 3, and 4 respectively.

Can declare the size of secondary dimension later

When declaring a two-dimensional array, it is not necessary to declare the size of the secondary dimension when the primary array is instantiated. Declaration of the size of each sub-array can be deferred until later as illustrated in this program.

Accessing an array out-of-bounds

This program also illustrates the result of attempting to access an element that is out-of-bounds. Java protects you from such programming errors.

ArrayIndexOutOfBoundsException

An exception occurs if you attempt to access out-of-bounds, as shown in the program in in Listing 1 (p. 6) .

In this case, the exception was simply allowed to cause the program to terminate. The exception could have been caught and processed by an exception handler, a concept that will be explored in a future module.

The program named array01

The entire program is shown in Listing 1 (p. 6) . The output from the program is shown in the comments at the top of the listing.

Listing 1: The program named array01.

```
    /*File array01.java Copyright 1997, R.G.Baldwin
Illustrates creation and manipulation of two-dimensional
array with the sub arrays being of different lengths.

Also illustrates detection of exception when an attempt is
made to store a value out of the array bounds.

This program produces the following output:

00
012
0246
Attempt to access array out of bounds
java.lang.ArrayIndexOutOfBoundsException:
     at array01.main(array01.java: 47)

*********************************************************/
class array01 { //define the controlling class
  public static void main(String[] args){ //main method
    //Declare a two-dimensional array with a size of 3 on
    // the primary dimension but with different sizes on
    // the secondary dimension.
    //Secondary size not specified initially
    int[][] myArray = new int[3][];
    myArray[0] = new int[2];//secondary size is 2
    myArray[1] = new int[3];//secondary size is 3
    myArray[2] = new int[4];//secondary size is 4
```

```
    //Fill the array with data
    for(int i = 0; i < 3; i++){
      for(int j = 0; j < myArray[i].length; j++){
        myArray[i][j] = i * j;
      }//end inner loop
    }//end outer loop

    //Display data in the array
    for(int i = 0; i < 3; i++){
      for(int j = 0; j < myArray[i].length; j++){
        System.out.print(myArray[i][j]);
      }//end inner loop
      System.out.println();
    }//end outer loop

    //Attempt to access an out-of-bounds array element
    System.out.println(
                  "Attempt to access array out of bounds");
    myArray[4][0] = 7;
    //The above statement produces an ArrayIndexOutOfBounds
    // exception.

  }//end main
}//End array01 class.
```

Assigning one array to another array – be careful

Java allows you to assign one array to another. You must be aware, however, that when you do this, you are simply making another copy of the reference to the same data in memory.

Then you simply have two references to the same data in memory, which is often not a good idea. This is illustrated in the program named **array02** shown in Listing 2 (p. 7) .

Listing 2: The program named array02.

```
    /*File array02.java Copyright 1997, R.G.Baldwin
Illustrates that when you assign one array to another
array, you end up with two references to the same array.

The output from running this program is:

firstArray contents
0 1 2
secondArray contents
0 1 2
Change a value in firstArray and display both again
firstArray contents
0 10 2
secondArray contents
0 10 2
**********************************************************/
class array02 { //define the controlling class
```

```
  int[] firstArray;
  int[] secondArray;

  array02() {//constructor
    firstArray = new int[3];
    for(int cnt = 0; cnt < 3; cnt++) firstArray[cnt] = cnt;

    secondArray = new int[3];
    secondArray = firstArray;
  }//end constructor

  public static void main(String[] args){//main method
    array02 obj = new array02();
    System.out.println( "firstArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.firstArray[cnt] + " " );
    System.out.println();

    System.out.println( "secondArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.secondArray[cnt] + " " );

    System.out.println();
    System.out.println(
      "Change value in firstArray and display both again");
    obj.firstArray[1] = 10;

    System.out.println( "firstArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.firstArray[cnt] + " " );
    System.out.println();

    System.out.println( "secondArray contents" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.print(obj.secondArray[cnt] + " " );

    System.out.println();
  }//end main
}//End array02 class.
```

5 Arrays of Objects

An array of objects really isn't an array of objects

There is another subtle issue that you need to come to grips with before we leave our discussion of arrays. In particular, when you create an array of objects, it really isn't an array of objects.

Rather, it is an array of object references *(or null)* . When you assign primitive values to the elements in an array object, the actual primitive values are stored in the elements of the array.

However, when you assign objects to the elements in an array , the actual objects aren't actually stored in the array elements. Rather, the objects are stored somewhere else in memory. The elements in the array

contain references to those objects.

All the elements in an array of objects need not be of the same actual type

The fact that the array is simply an array of reference variables has some interesting ramifications. For example, it isn't necessary that all the elements in the array be of the same type, provided the reference variables are of a type that will allow them to refer to all the different types of objects.

For example, if you declare the array to contain references of type **Object** , those references can refer to any type of object *(including array objects)* because a reference of type **Object** can be used to refer to any object.

You can do similar things using *interface* types. I will discuss interface types in a future module.

Often need to downcast to use an Object reference

If you store all of your references as type **Object** , you will often need to downcast the references to the true type before you can use them to access the instance variables and instance methods of the objects.

Doing the downcast no great challenge as long as you can decide what type to downcast them to.

The Vector class

There is a class named **Vector** that takes advantage of this capability. An object of type **Vector** is a self-expanding array of reference variables of type **Object** . You can use an object of type **Vector** to manage a group of objects of any type, either all of the same type, or mixed.

(Note that you cannot store primitive values in elements of a non-primitive or reference type. If you need to do that, you will need to wrap your primitive values in an object of a wrapper class as discussed in an earlier module.)

A sample program using the Date class

The sample program, named **array03** and shown in Listing 3 (p. 9) isn't quite that complicated. This program behaves as follows:

- Declare a reference variable to an array of type **Date** . *(The actual type of the variable is Date[].)*
- Instantiate a three-element array of reference variables of type **Date** .
- Display the contents of the array elements and confirm that they are all null as they should be. *(When created using this syntax, new array elements contain the default value, which is null for reference types.)*
- Instantiate three objects of type **Date** and store the references to those objects in the three elements of the array.
- Access the references from the array and use them to display the contents of the individual **Date** objects.

As you might expect from the name of the class, each object contains information about the date.

Listing 3: The program named Array03.

```
/*File array03.java Copyright 1997, R.G.Baldwin

Illustrates use of arrays with objects.

Illustrates that "an array of objects" is not really an
array of objects, but rather is an array of references
to objects.  The objects are not stored in the array,
but rather are stored somewhere else in memory and the
references in the array elements refer to them.

The output from running this program is:

myArrayOfRefs contains
```

```
null
null
null

myArrayOfRefs contains
Sat Dec 20 16:56:34 CST 1997
Sat Dec 20 16:56:34 CST 1997
Sat Dec 20 16:56:34 CST 1997
*******************************************************/
import java.util.*;

class array03 { //define the controlling class
  Date[] myArrayOfRefs; //Declare reference to the array

  array03() {//constructor
    //Instantiate the array of three reference variables
    // of type Date.  They will be initialized to null.
    myArrayOfRefs = new Date[3];

    //Display the contents of the array.
    System.out.println( "myArrayOfRefs contains" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.println(this.myArrayOfRefs[cnt]);
    System.out.println();

    //Instantiate three objects and assign references to
    // those three objects to the three reference
    // variables in the array.
    for(int cnt = 0; cnt < 3; cnt++)
      myArrayOfRefs[cnt] = new Date();

  }//end constructor
  //-----------------------------------------------------//

  public static void main(String[] args){//main method
    array03 obj = new array03();
    System.out.println( "myArrayOfRefs contains" );
    for(int cnt = 0; cnt < 3; cnt++)
      System.out.println(obj.myArrayOfRefs[cnt]);
    System.out.println();
  }//end main
}//End array03 class.
```

6 Strings

What is a string?

A string is commonly considered to be a sequence of characters stored in memory and accessible as a unit.

Java implements strings using the **String** class and the **StringBuffer** class.

What is a string literal?

Java considers a series of characters surrounded by quotation marks as shown in Figure 7 (p. 11) to be a string literal.

A string literal.

```
"This is a string literal in Java."
```

Figure 7: A string literal.

This is just an introduction to strings

A major section of a future module will be devoted to the topic of strings, so this discussion will be brief.

String objects cannot be modified

String objects cannot be changed once they have been created. If you have that need, use the **StringBuffer** class instead.

StringBuffer objects can be used to create and manipulate character data as the program executes.

6.1 String Concatenation

Java supports string concatenation using the overloaded + operator as shown in Figure 8 (p. 11) .

String concatenation.

```
"My variable has a value of " + myVar
   + " at this point in the program."
```

Figure 8: String concatenation.

Coercion of an operand to type String

The overloaded + operator is used to concatenate strings. If either operand is type **String** , the other operand is coerced into type **String** and the two strings are concatenated.

Therefore, in addition to concatenating the strings, Java also converts values of other types, such as **myVar** in Figure 8 (p. 11) , to character-string format in the process.

6.2 Arrays of String References

Declaring and instantiating a String array

The statement in Figure 9 (p. 12) declares and instantiates an array of references to five **String** objects.

Declaring and instantiating a String array.

```
String[] myArrayOfStringReferences = new String[5];
```

Figure 9: Declaring and instantiating a String array.

No string data at this point

Note however, that this array doesn't contain the actual **String** objects. Rather, it simply sets aside memory for storage of five references of type **String** . *(The array elements are automatically initialized to null.)* No memory has been set aside to store the characters that make up the individual **String** objects. You must allocate the memory for the actual **String** objects separately using code similar to the code shown in Figure 10 (p. 12) .

Allocating memory to contain the String objects.

```
myArrayOfStringReferences[0] = new String(
    "This is the first string.");
myArrayOfStringReferences[1] = new String(
    "This is the second string.");
```

Figure 10: Allocating memory to contain the String objects.

The new operator is not required for String class

Although it was used in Figure 10 (p. 12) , the **new** operator is not required to instantiate an object of type **String** . I will discuss the ability of Java to instantiate objects of type **String** without the requirement to use the **new** operator in a future module.

7 Run the programs

I encourage you to copy the code from Listing 1 (p. 6) , Listing 2 (p. 7) , and Listing 3 (p. 9) . Compile the code and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

8 Looking ahead

As you approach the end of this group of *Programming Fundamentals* modules, you should be preparing yourself for the more challenging ITSE 2321 OOP tracks identified below:

- Java OOP: The Guzdial-Ericson Multimedia Class Library [4]
- Java OOP: Objects and Encapsulation [5]

9 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Jb0240: Java OOP: Arrays and Strings
- File: Jb0240.htm
- Originally published: 1997
- Published at cnx.org: November 25, 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.

-end-

[4] http://cnx.org/content/m44148
[5] http://cnx.org/content/m44153