Lecture 05: Introduction to Arrays

This lecture contains:

- Fundamentals of Array Implementation in Java
- Array Manipulation
- Array of Objects
- Multi-dimensional Arrays
- Summary and Concluding Remarks
- Review Questions
- Recommended Readings

5.1 Fundamentals of Array Implementation in Java

As mentioned in lecture 1 (section 1.7.2), an array is a finite list of data items of a particular base type. Java implements, dynamic arrays, thus eliminating the need for pointers. The array can grow indefinitely. Further, unlike many programming languages (for example C, C++, Pascal, etc.), the Java array declaration does allocate memory, so there is no waste of space. This is so because Java separates the act of declaring an array from the act of allocating memory space for the array (you may combine the two actions but for maximum flexibility, it is recommended that you keep them separate).

Figure 5.1a provides the BNF syntax for declaring an array. Notice that when you declare an array, you do not specify its length as in many programming languages.

Figure 5.1a: Array Declaration in Java

```
ArrayDefinition ::=
<ArrayType>"[ ]"<ArrayName>;       // This is the preferred approach.

ArrayDefinition ::=
<ArrayType><ArrayName>"[ ]";       // This alternate approach has been included to satisfy C++
```

Once declared, you actually create the array (i.e. allocate memory space for it) via using the **new** operation as in following construct:

Figure 5.1b: Allocating Space for the Array

```
ArrayCreation ::=
<ArrayName> = new <ArrayType> "["<ArraySize>"]";
```

Please note the following:

1. The use of the square brackets in the (BNF) definition and creation is required; it does not indicate optional content as on other occasions; the use of the double quotation marks is intended to emphasize this.
2. The array-type (also called the *base type*) may be any valid primitive data type or class.
3. As mentioned above, Java makes a distinction between array declaration and array creation. The array declaration simply alerts the Java compiler that an array is to be used in that segment of the code. The actual creation can come later, when it is clear to the programmer, the required size of the array. It is at creation that the actual size (number of items) is specified.

Example 1: The following code declares float array, and subsequently creates it.

```
float [ ] salesList;   //Declares array of floating point numbers
…
salesList = new float [13];   //Creates the array with 13 entries
```

5.1 Fundamentals of Array Implementation in Java (continued)

Example 2: The two statements of Example 1 can be combined into one statement as in the following illustration. However, the approach of Example 1 is preferred because it is more flexible.

```
float [ ] salesList = new float [13];
```

Please note further:

1. The array creation actually creates an instance of a class. You can therefore access the properties of the array. One important property is the length of the array. You can access this property via the construct

   ```
   <ArrayName>.length
   ```

2. Array indexes (also called *subscripts*) start from 0 to <ArrayName>.lenght – 1. You can access a particular item view its index thus:

   ```
   <ArrayName> [IndexValue]  // Square bracket is required here also
   ```

5.2 Array Manipulation

You will soon find that there are countless scenarios where arrays will be applicable. Whenever you have to process a list of similar items, you are going to need the use of arrays. Having defined and created the array, you need to manipulate it. This section discusses array initialization and introduces you to basic array manipulation.

5.2.1 Array Initialization

Your array must be initialized with data items. This can be done via an initialization method, or via a shortcut declaration.

The shorthand approach is useful for small arrays (say fewer than ten items), where the array-type is a primitive data type. The array initialize notation is shown in figure 5.2.

Figure 5.2: Array Initialization for Trivial Arrays

```
ArrayInitialization ::=
<ArrayType> [ ] <ArrayName>= {<literal0>, <literal1> …};
```

5.2.1 Array Initialization (continued)

Note that the square bracket and curly braces are required in the syntax as shown. If your array is large (size of ten or more items), or the array type is not a primitive type, you need to write an initialization loop. Your initialization loop may be any iterative loop. The **For-Statement** is widely used for this purpose.

Example 3: To illustrate, figure 5.3 provides an initialization loop for the array of Example 1. This is the preferred approach for array initialization.

Figure 5.3: Illustrating Array Initialization via an Iterative Loop

```
float [ ] salesList;       //Declares array of floating point numbers
int listSize; // This will be used for sizing the array
// ...
salesList = new float [listSize];    //Creates the array with listSize entries
// ...
for (int x = 0; x < = salesList.length -1; x++) {salesList[x] = 0; }
```

5.2.2 Copying Arrays

The assignment

```
<Array 1> = <Array2>;
```

does not assign **Array2** to **Array1**. Rather, it assigns the reference of **Array 2** to the reference of **Array1**. This is so because an array is an object, not a primitive data item (review sections 4.8 and 4.9 of the previous lecture). To do an element by element copy, you need to write a loop or use the static **arrayCopy** method in the **System** class. The **System.arrayCopy** method has the following signature (remember, a method signature is simply the heading of the method):

```
public static void arrayCopy(Object [ ] sourceArray, int sourceStartPos, Object [ ] targetArray, int  +
targetStartPos, int length)
```

The **sourceArray** represents the array from which data is to be copied. The **sourceStartPos** represents the starting subscript in the source array. The **targetArray** represents the array to be copied to (i.e. the destination array). The **targetStartPos** represents the starting subscript in the destination array. The **length** represents the number of items to be copied from the source array to the target array. The reason an array of **Object** is used is due to the fact that **Object** is the supreme super-class in Java. All data items are therefore by definition, implicit instances of **Object** (the **Object** class will be further discussed in lecture 6).

Example 4: Figure 5.4 illustrates how you may copy all the elements of one array to another, using either the **arrayCopy(…)** method, or an iterative loop.

Figure 5.4: Copying Arrays

```
float [ ] salesList1, salesList2;          int numSales;
// ...
salesList1 = new float [numSales];    salesList2 = new float [numSales];
// ...
for (int x = 1; x < = numSales; x++) {salesList1[x-1] = salesList2[x-1]; } // Copies salesList2 to salesList1
```

```
float [ ] salesList1, salesList2;          int numSales;
// ...
salesList1 = new float [numSales];    salesList2 = new float [numSales];
// ...
System.arrayCopy(salesList2, 0, salesList1, 0, numSales);  // Copies salesList2 to salesList1
```

5.2.3 Passing Arrays to and from a Method

You can define a method to receive an array as a parameter, by simply declaring the array as parameter in the method heading. Of course, more than one array may be specified as parameters of the method. When the method is called, an array must be supplied as the argument in the usual way.

Let us revisit figure 5.4 above. In the figure, the static method **System.arrayCopy** is called with two arrays as arguments, namely **salesList1** and **salesList2**. Additional arguments relate to the starting position in each array and the number of items to be copied as clarified above.

You can also return an array from a method. The return type from the method would be of the following format:

```
<ArrayType> [ ]
```

Example 5: In the following code, a static method called **analyzeList(…)** accepts an array as argument processes it and returns the array to the calling statement.

Figure 5.5: Illustrating a Method that Receives an Array, and Returns it to the Calling Statement

```
public static float [ ] analyzeList (float [ ] thisList)
{
   // Within this method, the array is processed
   …
   return thisList; // The array is returned
}
```

Please note: The array(s) passed to a method may be completely different (not just in content, but name and base type) from the one returned from the method. Also, a method may receive array(s) but not return any, or it may not receive array(s) but returns an array. The illustration above simply represents the two scenarios of receiving an array, and returning an array.

5.2.4 Finding the Smallest and Largest Item in a List

Finding the smallest item and/or largest item in a list is a very common problem that people have to solve. For instance, a school teacher or college professor may want to know the best and worst performance from a class on a particular assignment. Figure 5.6 illustrates a generic algorithm for solving the problem. Assuming the declarations of Example 1, figure 5.7 shows a Java implementation of the algorithm.

Here are a few points to note about the algorithm and its Java implementation:

1. The algorithm receives an array of data items of a given base type. This array is called **thisList**.
2. The variable or object **lowes**t is initialized to the highest possible value. This is so because on each iteration through the loop, it will be compared with each item and switched if necessary.
3. The variable or object **highes**t is initialized to the lowest possible value. This is so because on each iteration through the loop, it will be compared with each item and switched if necessary.
4. Depending on the **highLowFlag**, the algorithm returns the highest or lowest item in the list.

Figure 5.6: Finding the Largest and Smallest Item in a List

```
Algorithm: highLow(thisList, thisLength, highLowFlag) Returns data item of the base-type of thisList
// Assume thisList be an array of any given base-type;
// Assume thisLength an integer;
Let x be an integer;
Let highOrLow be a variable of the base-type of thisList;
Let highLowFlag be a character; // 'H' represents high, 'L' represents low
Let lowest be a variable of the base-type of thisList, initialized to the highest possible value;
Let highest be a variable of the base-type of thisList, initialized to the lowest possible value;

START
  If (highLowFlag = 'H')
     For (x varying from 1 to thisLength with increments of 1) do the following:
          If (thisList[x - 1] > highest)
            highest :=  thisList[x - 1];
          End-If
     End-For;
     highOrLow := highest;
  End-If;

  If (highLowFlag = 'L')
     For (x varying from 1 to Length with increments of 1) do the following:
          If (thisList[x - 1] < lowest)
            lowest :=  thisList[x - 1];
          End-If
     End-For;
     highOrLow := lowest;
  End-If;

  Return highOrLow;
STOP
```

Figure 5.7: Java Implementation of the highLow Algorithm for a list of Real Numbers

```
public static float highLow(float [ ] thisList, char hghLowFlag)
{
  int x;
  float highOrLow, Highest = Float.MIN_VALUE, Lowest = Float.MAX_VALUE;

  if (highLowFlag == 'H')
  { for (x = 1, x <= thisList.length; x++) if (thisList[x-1] > Highest) Highest = thisList[x-1];
    highOrLow = Highest;
  }

  if (highLowFlag == 'L')
  { for (x = 1, x <= thisList.length; x++) if (thisList[x-1] < Lowest) Lowest = thisList[x-1];
    highOrLow = Lowest;
  }

  return highOrLow;
}
```

There is a more efficient way of finding the smallest or largest item in a list; it involves the use of a more advanced algorithm called a binary search. However, we will forego that discussion at this time; you will likely learn about binary search in a more advanced course such as Data Structures and Algorithms.

5.3 Array of Objects

As mentioned in earlier, the base-type of an array may be any valid primitive type or class. It follows therefore that you can declare an array of objects of a known instance class. In fact, arrays of objects are far more interesting than arrays of primitive data items.

When working with arrays of objects, you must be careful to remember that objects are reference variables. You should therefore not confuse the use of the assignment operator with invoking the appropriate setter method of the objects class to change its content. As a general rule, when manipulating objects, remember the following:

- Always instantiate the object before using it.
- To change an object's content, use the appropriate setter method defined in the object's parent class.
- To retrieve information about an object, use the appropriate getter method defined in the object's parent class.
- Never attempt to assign an object to another via the assignment operator. In more advanced programming you may make such assignment; however, such manipulations are beyond the scope of an introductory course as this is, so do not concern yourself with such matters at this level.

Example 6: The following code illustrates (figure 5.8) how you may define an array of student records (objects). The assumption is that before doing this, a class called **StudentClass** would be defined.

Figure 5.8: Illustrating Definition of an Array of Student Objects

```
//Let StudentClass be a class of student objects. We may declare an array of student objects as follows:

StudentClass [ ] studList; int sLimit;
 // ...
 studList = new StudentClass[sLimit];
```

```
//Or simply:
StudentClass []studList = new StudClass[sLimit];
```

Example 7: Let us revisit the **LibraryPatron** class of the previous lecture (section 4.5). Suppose that we want to maintain a list of library patrons for various purposes. For instance, we may want to find the name of all patrons whose major is Computer Science, Biology, Mathematics, or some other field; or we may want to find out all the patrons with a bad library status; or we may be interested in a particular patron; and so on. Before we can provide all these services, we must let our program accept information for various library patrons and store them in an array. Figure 5.9a repeats the UML class diagram for the **LibraryPatron** class, and figure 5.9b shows the UML diagram for a driver class called **LibraryPatronsDemo02**, which will manipulate an array of **LibraryPatron** objects. Finally, the actual Java code for the driver class is provided in figure 5.9c. Here is a summary of what the program does:

1. Provide the user with a menu of three options: to enter information on library patrons; to query a particular library patron of choice; exit.
2. If the user takes option 1, he/she is prompted to enter the information for the library patrons. The information entered is stored in an array.
3. If the user takes the second option, he/she is prompted to specify the **pNumber** of the library patron. This is used to search the array and display information for that patron on screen. If an invalid number is specified, an error message is printed on screen.
4. The third option causes on exit from the program.

Additional Exercise: Study the figures 5.9a – 5.9c carefully and try to add additional features to the **LibraryPatronsDemo02** class. For instance, you may include a method for listing the contents of the array.

So as you can see, the effective use of arrays is critical to good programming. In fact, you cannot get very far without using them. At this point you may be wondering, what if I want to save the library patron records (objects) that I have received into my array, so I can use them at a later date? If you did, then that is good thinking. Arrays are only useful for holding *transitory data*. By this, we mean data that exist during the execution of a program, but are nonexistent thereafter. To address your question, we need a mechanism for storing *persistent data* i.e. data that exist and remain valid after the execution of a program. To store and access persistent data, you need to learn how to manage files (text files and binary files). You will do so later in your programming journey (part 2 of this course). At that point, you will learn how to read files into arrays and write files from arrays.

Figure 5.9a: The UML Diagram of the LibraryPatron Class

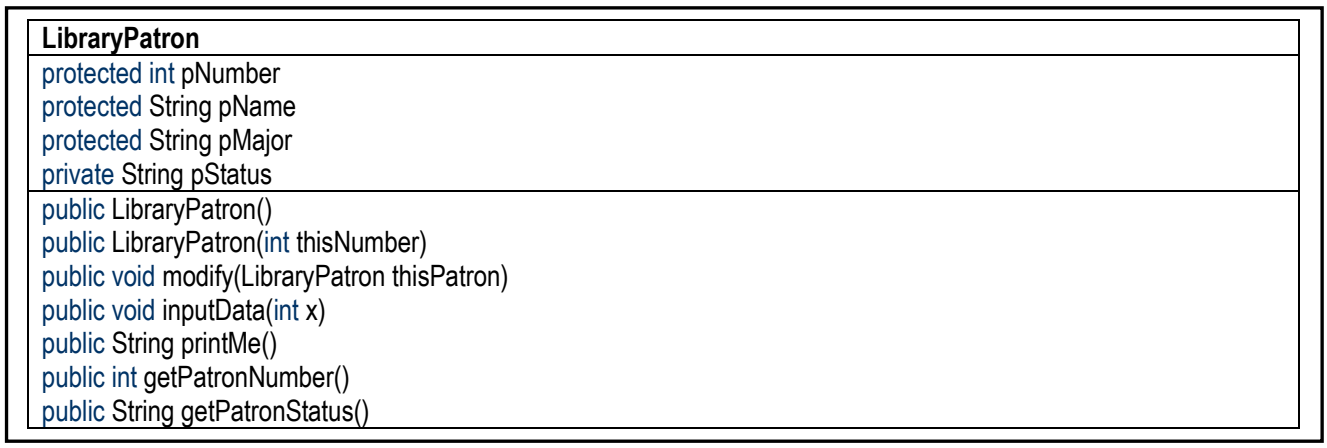

Figure 5.9b: The UML Diagram of the LibraryPatronDemo02 Class

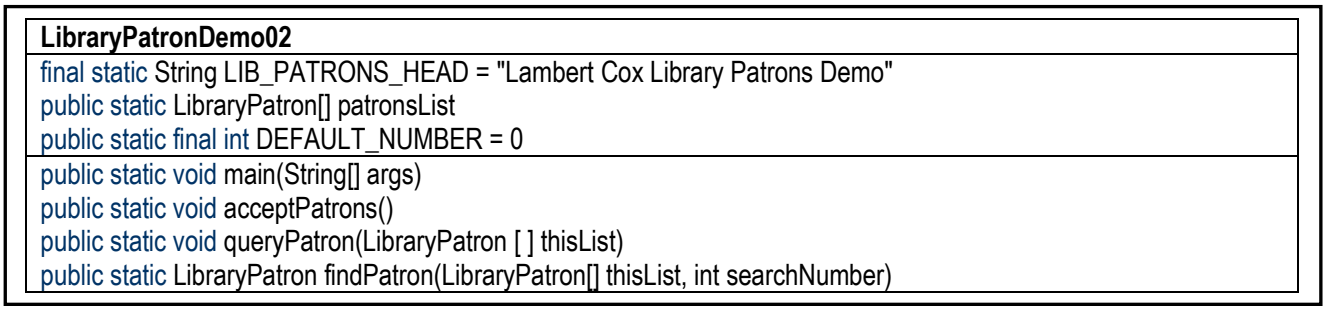

Figure 5.9c: The LibraryPatronsDemo02 Class

```
// LibraryPatronDemo02.java:  Allows for the entry of Library Patron objects, and storage in an array.
// Also allows user to search for a particular patron and display the information.
// Author Elvis Foster
// ***************************************************************************
package Application3;
import javax.swing.JOptionPane; // This package facilitates dialog boxes, etc.

public class LibraryPatronsDemo02
{
   public static LibraryPatron[] patronsList;
   public static String inputString;
   public static final String LibPatronsHEAD = "Library Patrons Demo 2";
   public static final int DEFAULT_NUMBER = 0;
   //LibraryPatron CurrentPatron, Dummy;

   /* No constructor required
   public LibraryPatronsDemo02() { } */

   // Main Method
   public static void main(String[] args)
   {
    // Declare Variables
    final String promptString = "1. Add Library Patrons \n" + "2. Find a Library Patron \n" + "9. Exit Program \n";
    boolean exitTime = false;
    char exitKey = ' ';
    int userOption;

    // Main Loop
    while (!exitTime)
    {
     // Present the menu
     inputString = JOptionPane.showInputDialog(null, promptString, LibPatronsHEAD, JOptionPane.QUESTION_MESSAGE);
     userOption = Integer.parseInt(inputString);

     // Respond to the request
     switch (userOption)
     {
       case 1: {acceptPatrons(); break;}
       case 2: {queryPatron(patronsList); break;}
       case 9: {exitTime = true; break;}
     }

     // Check whether user wishes to continue
     if (!exitTime)
     { inputString = JOptionPane.showInputDialog(null, "Press X to exit, or any other key: ", LibPatronsHEAD,  +
          JOptionPane.QUESTION_MESSAGE);
       exitKey = inputString.charAt(0);
       if (exitKey == 'X' || exitKey == 'x') exitTime = true;
     }

    } // End While
   } // End main
```

Figure 5.9c: The LibraryPatronsDemo02 Class (continued)

```
// acceptPatrons Method
public static void acceptPatrons()
{
    int numPatrons, x;
    inputString = JOptionPane.showInputDialog(null, "Number of Patrons: ", LibPatronsHEAD, JOptionPane.QUESTION_MESSAGE);
    numPatrons = Integer.parseInt(inputString);

    patronsList = new LibraryPatron [numPatrons]; // Creates an array of LibraryPatron objects
    for (x =1; x <= numPatrons; x++)
     {
       patronsList[x-1] = new LibraryPatron();  // patronsList[x-1].modify(Dummy) is logically incorrect;
       patronsList[x-1].inputData(x); // Prompt For and Accept LibraryPatron Data
     };   // End For
 } // End of acceptPatrons Method

 // The queryPatron Method
 public static void queryPatron(LibraryPatron [ ] thisList)
 {
   /* Note: The approach used to test for object equality is based on the PatronNumber for LibraryPatron.
    A more elegant approach would have been to use an overrided form of the equals method from the Object class  */

   // Declarations
   String searchArg, outString;
   int searchNumber;
   String heading = "Library Patron Query";
   LibraryPatron foundPatron;
   boolean exitNow = false;

   // Prompt for the Patron's Number; then use it to find the patron
   if (thisList != null) // If there are patrons
   {
     while (!exitNow) // While more processing required
     {
       searchArg = JOptionPane.showInputDialog(null, "Specify the Patron Number of Interest: ", heading, +
           JOptionPane.QUESTION_MESSAGE);
       searchNumber = Integer.parseInt(searchArg);
       foundPatron = new LibraryPatron();
       foundPatron.modify(findPatron(thisList, searchNumber));

       // Output the result of the search
       if (foundPatron.getPatronNumber() != DEFAULT_NUMBER)outString = foundPatron.printMe();
       else outString = "Library patron specified is not in the list.";
       JOptionPane.showMessageDialog(null, outString, heading, JOptionPane.INFORMATION_MESSAGE);

       // Prompt user whether to continue or not
       String More = JOptionPane.showInputDialog(null, "Press any key to continue or X to exit Library Patron Query", heading, +
           JOptionPane.QUESTION_MESSAGE);
       char exitKey = More.charAt(0);
       if (exitKey == 'X' || exitKey == 'x') exitNow = true;

     } // End-While more processing required
   } // End-If there are patrons
   } // End QuerySalesman

// ...
```

Figure 5.9c: The LibraryPatronsDemo02 Class (continued)

```
// The findPatron Method
   public static LibraryPatron findPatron(LibraryPatron[] thisList, int searchNumber)
   {
     // Declarations
     LibraryPatron Found = new LibraryPatron();
     boolean exitNow = false;

     // Search ThisList for the LibraryPatron object, then return it
     for (int x = 1; ((x <= thisList.length) && (!exitNow)); x += 1)
     {
       if (searchNumber == thisList[x-1].getPatronNumber())
       {
         Found.modify(thisList[x-1]);
         exitNow = true;
       }
     } // End For
     return Found;
   } // End findPatron

} // End of LibraryPatronsDemo02
```

5.4 Multi-dimensional Arrays

Java allows you to define and work with multi-dimensional (MD) arrays. In fact, the arrays that we have been working with so far are one-dimensional (1D) arrays (also called vectors). To declare an MD array, you introduce a pair of square brackets for each dimension (see figure 5.10a). As for creating the MD array, you use the new operator as depicted in figure 5.10b, including a square bracket for each dimension.

Figure 5.10a: Declaring Multi-dimensional Arrays

ArrayDefinition ::=
<ArrayType>[ ] … [ ] <ArrayName>; // This is the preferred approach.

ArrayDefinition ::=
<ArrayType><ArrayName>[ ] … [ ]; // This approach has been included to satisfy C++ programmers.

Once declared, you actually create the array using the following construct:

Figure 5.10b: Creating Multi-dimensional Arrays

ArrayCreation ::=
<ArrayName> = new <ArrayType> "["<Dim1Size>"]" … "["<DimNSize>"]";

Here, **Dim1Size** represents the size (maximum length) for the first dimension, and **DimNSize** represents the size for the n^{th} dimension.

5.4 Multi-dimensional Arrays (continued)

Example 8: The following example (figure 5.11) declares and creates a 2D array — **salesMatrix** — of floating point numbers. This array could be used to represent the sale of **sMax** (for instance 20) salesmen over a period of **mMax** (for example 12) months, with a 13th period for the end of the financial period. Each row could represent the performance of a particular salesman. The subscripts will range from **saleMatrix**[0] [0] to **saleMatrix**[sMax -1] [mMax - 1].

Figure 5.11: Illustrating a 2D Array to Track Sales

```
String [ ] salesAgentList;
float [ ][ ] salesMatrix;
int sMax, mMax;

// ...
salesMatrix = new float [sMax] [mMax];
salesAgentList  = new String [sMax];

// Alternate declaration of the SaleMatrix follows:
float [ ][ ] salesMatrix = new float [sMax] [mMax]; // Merges the two statements into one statement.
```

Example 9: Assuming the sales matrix of Example 8, figure 5.12 provides an initialization method, followed by a method for computing the average sale of a given sales agent.

Figure 5.12: initializeSaleMatrix Method & findAverage Method for saleMatrix of Example 8

```
// Initialization method
// Assume that the declarations of figure 5.11 are applicable here
public static float [ ] [ ] linitializeSaleMatrix (float [ ] [ ] thisList, int sMax, int mMax);
{ int salesAgent, month;
   for (salesAgent = 1; salesAgent <= sMax; salesAgent ++)
    { for (month = 1; month <= mMax; month++) thisList[salesAgent -1] [month-1] = 0.0; }
   return thisList;
}; // End of Initialize method

// findAverage method: Calculates and returns the average sale of a sales agent
public static float findAverage (float [ ] [ ] thisList, int thisSalesAgent, int mMax);
{ float totalSale = 0.0, averageSale = 0.0;
   for (int month = 1; month <= mMax; month++) totalSale += thisList [thisSalesAgent-1] [month-1];
   averageSale = totalSales / mMax;
   return averageSale;
}  // End of findAverage method
```

5.5 Summary and Concluding Remarks

An array is a finite list of data items of a particular base type. Java implements dynamic arrays, thus eliminating the need for pointers. The array can grow indefinitely. Further, unlike many programming languages, the Java array declaration does allocate memory, so there is no waste of space. This is so because Java separates the act of declaring an array from the act of creating the array (you may combine the two actions but for maximum flexibility, it is recommended that you keep them separate).

The recommended way to initialize an array is to use an initialization loop. Your initialization loop may be any iterative loop; the **For-Statement** is widely used for this purpose. The alternate shortcut initialization statement is recommended only for trivial cases where the array consists of a few occurrences of a primitive data item.

To copy an array to another, use an iterative loop and copy the items one-by-one, or use the **arrayCopy(…)** method.

A method can have multiple arrays as input parameters; additionally, a method can return an array to the calling statement.

The following strategy is recommended for finding the smallest item in a list:
- Define a data item denoted as **lowest**, of the same data type as the list's base type.
- Initialize **lowest** to the largest possible value in the list.
- Iterate through the list, and on each iteration, compare **lowest** to the current item. If the value of the current item is smaller than **lowest**, then assign its value to **lowest**.
- Emerging from the iteration loop, **lowest** will contain the value of the smallest item in the list.

The following strategy is recommended for finding the largest item in a list:
- Define a data item denoted as **highest**, of the same data type as the list's base type.
- Initialize **highest** to the smallest possible value in the list.
- Iterate through the list, and on each iteration, compare **highest** to the current item. If the value of the current item is larger than **highest**, then assign its value to **highest**.
- Emerging from the iteration loop, **highest** will contain the value of the largest item in the list.

Since the base-type of an array may be any valid primitive type or class, it therefore follows that you can declare an array of objects of a known instance class. While arrays of objects are far more interesting than arrays of primitive data items, in manipulating the former, you must remember that objects are treated differently from the way primitive data items are treated. Essentially, you manipulate objects by working through their manipulators — getters and setters defined in the object's parent class.

Java allows you to define and work with multi-dimensional (MD) arrays. In fact, the arrays that we have been working with so far are one-dimensional (1D) arrays (also called vectors). To declare an MD array, introduce a pair of square brackets for each dimension. As for the one-dimensional (1D) array, you declare an MD array in one of two ways, as depicted in figure 5.10.

As your knowledge and confidence in programming grows, you will doubt find countless occasions to use arrays. Moreover, in time, you will come to understand that the use of arrays is commonplace in programming at the intermediate and advanced levels.

5.6 Review Questions

1. What is an array?

2. Briefly describe how Java handles arrays.

3. Describe two strategies for initializing an array in Java. When would you use each strategy?

4. Write a Java program to maintain four lists, each of the same size as specified by the user. List 1 should store double precision numbers keyed in by the user. List 2 should be automatically constructed from list 1, where each item is larger than the corresponding item in list 1 by a factor specified by the user. List 3 should be automatically constructed by calculating the difference between corresponding items in list 1 compared to list 2. For instance, if x is a subscript, then **list3**[x] stores **list2**[x] – **list1**[x]. List 4 should be automatically constructed by calculating the sum of corresponding items in list 1 and list 2. For instance, if x is a subscript, then **list4**[x] stores **list2**[x] + **list1**[x]. Your output should display the contents of each list.

5. What precautions should be followed when manipulating an array of objects?

6. Rewrite the program from review question 8 of the previous chapter, this time using an array of student objects to facilitate efficient processing. The problem is restated below:

   Construct a UML class diagram for an instance class to keep track of students at a college. Suggested data item to track are student identification number, name, telephone number, date of birth, first major, second major or minor, and grade point average (GPA). Be sure to include the essential manipulators in the diagram. Consider your class diagram and then respond to the following questions/instructions:

   What data items would you perform data validation on?

   Write appropriate Java code to implement your **Student** class.

   Construct a second UML diagram to represent a driver class that manipulates instances of your **Student** class.

   Write appropriate Java code for your driver class. For instance, you may do the following:
   - Accept and process a list of Student instances (consecutively).
   - Determine the **Student** instance with the highest GPA as well as the **Student** instance with the lowest GPA.
   - Calculate the average GPA for the batch of **Student** instances.
   - Output the results based on the analysis.

7. Briefly describe how multi-dimensional arrays are declared, constructed, and manipulated in Java.

5.7 Recommended Reading

[Bell & Parr 2010] Bell, Douglas and Mike Parr. 2010. *Java for Students* 6th Ed. New York: Pearson. See chapters 13 & 14.

[Liang 2014] Liang, Y. Daniel. 2014. *Introduction to Java Programming — Comprehensive Version*, 10h ed. Boston, MA: Pearson Education. See chapters 7 & 8.

[Savitch & Carrano 2008] Savitch, Walter and Frank M. Carrano. 2008. *Java: An Introduction to Problem Solving & Programming* 5th ed. Upper Saddle River, NJ: Prentice Hall. See chapter 7.

[Oracle 2015]. Oracle Corporation. 2015. "Java™ Platform, Standard Edition 8API Specification." Accessed January 19, 2015. http://docs.oracle.com/javase/8/docs/api/