**Chapter X**

**Java Static 1D & 2D Arrays**

**Chapter X Topics**

10.1 Introduction to Data Structures

10.2 Array Definition

10.3 1D Array Declaration and Access

10.4 Random Arrays

10.5 Accessing Array Elements with the <for..each> Loop Structure

10.6 Introduction to Static 2D Arrays

10.7 Controlling 2D Array Output

10.8 Storing Objects in a Static Array

10.9 2D Arrays and Length

10.10 Parameter Differences Between Simple Data Types and Arrays

10.11 Summary

**10.1 Introduction to Data Structures**

Early in the course you were introduced to simple data types, like **int**, **float**, **char**, **double**, and **boolean**. Each of these data types can be used to create a variety of required variables. A simple data type variable is a location in memory that stores a **single** value that can be used by a computer program. Single values are practical for loop counter variables, maximum number of grades, the height of *Pikes Peak* and the number of medals won by the United States at the last Olympics. Programs that handle passenger airline reservations, student college transcripts, employee payroll records and hospital patient information, require massive data storage. Such major storage requirements cannot be handled efficiently by thousands of simple data type variables, each storing a single value. You will need more sophisticated data types.

It can be argued that you have been storing multiple values inside objects since the very beginning, and that is very true. However, the data stored inside the classes and objects so far have been one or more simple data types. There are many situations where data needs to hold more than one value. Such a situation calls for using a **data structure**. So what is a data structure? Look at a building. Note that it is made up of smaller structures like rooms, halls, stairways, etc. A room is made up of walls, floors, ceilings, desks, chairs, etc.

Another example can be found with animals. Animals are organisms made up of *organ systems*. Each organ system is made up of *organs*. Organs are made up of *tissues*, and tissues are made up of *cells*. We could continue and work down to the molecular and atomic level, but for this analogy, assume that the **cell** is the simplest, lowest level. The whole point is that the structure, an organism in this case, is made up of other, smaller structures, until eventually you reach the smallest component.

These two examples are used to motivate the definition of a data structure. In computer science it really is the same idea. The only difference in structures is the nature of the smallest building block used to create the structure. In an animal organism it is a cell. In a building it may be a brick or a plank and in a computer science data structure it is a simple data type.

|  |
| --- |
| **First Data Structure Definition** |
| A **data structure** is a data type whose components are  smaller data structures and/or simple data types. |

You will note that it states *First Data Structure Definition*. This definition is not quite finished. We will revisit the data structure definition again and make some revisions. The complete, and more accurate, definition will only add unnecessary complexity right now. First we need to spend some time with a variety of data structure examples before it makes sense to become more precise. This approach is quite similar to teaching somebody to play a new card game. It just does not make sense to explain all the more intricate details of the game when playing for the first time. Frequently, the best approach is to deal the cards and explain as you go along. After several hands are dealt, it is easier to summarize a variety of rules. In other words, let us deal some hands first and then we talk some more.

So what is the bottom line essence of this data structure, right now at this early stage? It is simple. You no longer have a data type that stores a single value. You can store more than one value in a data type that is a data structure. Put in other words, any data type that can store more than one value is a data structure.

|  |
| --- |
| **Data Structure Starting Point** |
| Any data type that can store more than one value is a data  structure. |

Alright, we have a starting point. Now we need to look and see what computer science has to offer for us in the data structures department. Exactly, what types of data structures exist and what can they do for us. You may have noticed that the title of this chapter talks about *arrays*, which is one kind of data structure. The importance of data structures is such that one chapter is devoted to each data structure. Since this is the very first chapter about any kind of data structure, it will help to give a brief overview of several different types of data structures.

**The Array Data Structure**

The **array**, or **subscripted variable**,is the first historical data structure. This data structure became popular with the use of the first commercially, widely-used, programming language, **FORTRAN**. FORTRAN, which means FORmula Translator, was designed for the scientific - number crunching - community. A data structure, like an array, was necessary for the storing and processing large quantities of numbers.

What does an array bring to mind? How about an array of flowers or an array of books, or an array of anything else? We think of an array as having multiple items - not a single item - and an array has the same type of items. We can have an array of integers, an array of real numbers, an array of characters, and an array of strings. An array can have any kind of element, as long as each element is the same data type. You will find the name *vector*used frequently for *one-dimensional* arrays and *matrix*for *two-dimensional*arrays.

|  |
| --- |
| **First Array Definition** |
| An **array** is a data structure with one, or more, elements  of the same type.  A one-dimensional array is frequently also called a **vector**.  A two-dimensional array is frequently also called a **matrix**. |

Once again you see this *first definition* expression. This is the same story as the data structure definition. All this business is tied together. More complete definitions will come later when each data structure is explained in more detail.

**The Record Data Structure**

The business community was not very happy with the FORTRAN language and particularly with the data structure limitation of having an array and nothing else. In the business world, data is not of the same type. This is lovely in science and math where numbers rule the discipline, but in business it is another story.

Data storage in business requires storing names, addresses, birth dates, number of dependents, social security numbers, credit cards numbers, flight numbers, years worked, pay rates, credit balance available, etc. etc. etc. One solution was to create many different arrays. Each array specialized in one part of some business record. An array of names, an array of addresses, an array of pay rates and so on, which were called *parallel arrays*. This worked, but it was tedious.

A new programming language became popular, called COBOL (COmmon Business Oriented Language), which introduced the **record** data structure. What does the word **record** bring to mind? How about a student’s record, an employee’s record, a patient’s record, a passenger’s record? Each one of these records has the common thread of multiple information fields that can be of many different types.

This type of data structure is precisely what the business world required. COBOL became a highly successful language (it helped that the Department of Defense adopted the language) and the record is now an integral part of programming. Initially, records only stored data of different types. Over time, records have evolved and now improved records store actions that process the data inside the same data structure. You have seen this improved record structure already in Java as the **class**, which is the primary component of Object Oriented Programming.

|  |
| --- |
| **Record Definition** |
| A **record** is a data structure with one, or more, elements,  called *fields*, of the same or different data types. |

**The File Data Structure**

Programming languages have a convenient data structure that facilitates the transfer of data to and from external storage. The array and record may be lovely to store a complex set of data in the memory of a computer, but this data often needs to be used at some future date. The data needs to be stored in some permanent manner. The file data structure provides this ability.

|  |
| --- |
| **File Definition** |
| A **file** is an *internal* data structure - with an unspecified  number of elements of the same type - assigned to an  *external* file name. The file data structure allows transfer  of data between internal and external storage. |

**Other Data Structures**

The three data structures - array, record and file - introduced in this section are built-in Java data types. These data types are ready to go and can be used with very little effort. Using built-in data structures is a good starting point in an introductory computer science course. There are many other types of data structures that the programmer can create for a wide variety of special purposes. The study and practice of these special user-created data structures is a major focus of the second year AP Computer Science course. One example of such a special data structure will be presented here, the **stack**.

**The Stack Data Structure**

One important data structure in computer science is the **stack**. This data structure will be explained, and used, in considerable detail in the future. Right now consider the following stack definition.

|  |
| --- |
| **Stack Definition** |
| A stack is a data structure with elements of the same type.  Data elements of the stack data structure can only  be accessed (stored or retrieved) at one end of the stack  in a **LIFO** (Last In, First Out) manner. |

Let us go to a large home improvement store to create an analogy about this data structure business. You walk through one isle to get a variety of bolts and nuts of different sizes. All the hardware is neatly organized in separate containers that are clearly labeled. You might think that the isle containing all the screws, bolts, nuts and hooks is a large record of hardware data. There is one organized isle for many different types of hardware. You walk directly to a container that is marked with the size bolt that you need. After that you walk to another container that stores the correct sized nut that you need. This is **random access**. You can select to access any items in any random pattern.

A little later you need to pick up a new lawnmower. All the new lawnmowers are stored in neat stacks, eight boxes high. The particular lawnmower that you need happens to be the third box from the bottom of one stack. It is not possible for you to access this lawnmower randomly. The stack only allows access at the top. Store employees carefully remove one box at a time with a forklift from the top of the stack, until your lawn mower can be accessed.

This is **not random access**. Data access to the lawnmowers or a computer science stack is only possible at one end. Furthermore, the access is **Last In, First Out** (**LIFO)**.

Now why do you need to use a stack data structure in computer science? This is a topic for later discussion. Right now you need to learn about arrays. The forward peek to the stack was provided to make a relevant comparison of different data access. It was used strictly to help explain that the manner of data access is fundamental to a data structure's definition.

The understanding and use of data structures is one of the most significant components of successful programming. You will be using many data structures, both Java provided data structures, and user-created data structures.

The definition of a data structure, given at the beginning of this introduction -- and this has been a long introduction -- will be repeated here. Look at this short sentence closely. The definition is strictly limited to the storing of information. Nothing is stated about how the information accessed.

|  |
| --- |
| **First Data Structure Definition** |
| A **data structure** is a data type whose components are  smaller data structures and/or simple data types. |

This is precisely why this is called *First Data Structure Definition*. The definition was fine for the very first introduction of a data structure, but it is not complete. There is more to the story. Something has to be mentioned about the manner in which data is accessed.

Let us make an analogy with a car here. A car is a complex piece of machinery that consists of many components. We can somewhat think of a car as a data structure with components like doors, lights, transmissions, radios, steering wheels, etc.

It is not sufficient to define a car by specifying that the car has doors, lights, a transmission, a radio, a steering wheel, etc. The access of these components is a major part of the overall definition or understanding of the car.

Do the doors only open with a key, or does it have remote access, or perhaps a combination code that must be entered to unlock the doors? Is the transmission automatic, or manual, and if it is manual, how many gears are there and what is the pattern? Furthermore, is the transmission two-wheel drive or four-wheel drive? The steering wheel controls the direction of the car, but is it direct access or is it indirect access with power steering?

These are all questions that must be answered before somebody purchases a car. The precise definition of a car cannot be summed up by its components. The manner in which the components are accessed or operate has to be part of a complete definition.

The same is true with data structures. Yes, we need to know how data is stored and what type of data can be stored. One major difference between an **array** and a **record** is the fact that an array can only store data of the same type, while the record can store data of many different types. That is great, but it is not sufficient. How do we access the data and what can be done to the data is another question? Consider the following altered data structure definition.

|  |
| --- |
| **Improved Data Structure Definition** |
| A **data structure** is a data type whose components are  smaller data structures and/or simple data types. The  storing and retrieval of the data elements is performed by  accessing methods that characterize the data structure. |

All of a sudden the clean and simple data structure definition has become rather nasty looking. Hopefully, it will not seem all that bad. The first sentence is old stuff. We have been there before talking about data access. The second sentence explains that data has to be accessed and the manner in which data is accessed and processed defines the nature of the data structure.

The remainder of this chapter will concentrate completely on arrays or subscripted variables. The long introduction was provided to get some basic feel about the nature of data structures. Do not be concerned if you know little about arrays. The purpose of this chapter is precisely to clarify the array data structure. Right now it is hoped that you have some feel for data structures in general

**10.2 Array Definition**

What comes to mind when you think of an *array*? There is an *array* of flowers, and you may also have an *array* of Barbie dolls, or perhaps an *array* of kitchen pots and pans. In each case the array has a dual meaning. You are talking about more than one element. And you are also indicating that these elements are alike. They do not all need to be identical, but they are of an identical type. The array of flowers may include many different flowers, but they are all flowers.

If we only consider data storage, the following array definition is quite adequate. The definition explains that an array is a data structure, and it explains that the multiple elements of the array are fixed in number and they are of the same type. This is the definition that was presented in the introduction, a few pages back.

|  |
| --- |
| **First Array Definition** |
| An **array** is a data structure with a fixed number of elements  of the same type. |

Data structures are more than a means to store data. That was the main point made in switching from the first data structure definition to the improved data structure definition. The way in which the stored data is accessed is part of the data structure definition. This really is the essence of OOP *encapsulation*. Store both the data and the actions that access the data. This means that the first array definition is not complete. Some indication must be given about data access. Do not get too excited because Array data access is quite simple.

|  |
| --- |
| **Improved Array Definition** |
| An **array** is a data structure with a fixed number of elements  of the same type. Every element of the array can be  accessed directly. |

The improved array definition indicates that arrays can be accessed directly. How is this accomplished if there are many different values stored in the same data type? Arrays use some unique index or subscript to identify each element in the data structure. The indexing approach is all around us. Streets are arrays of homes. It is not sufficient to state that you live on *Main Street*. You need something like ***1750*** *Main Street*. An airplane contains an array of seats, but *flight 512*specifies the flight, not the location where you sit in the plane. A boarding pass will say something like *Flight 512,* ***Row 32****,* ***Seat D***. A similar system applies to a football game’s reserved seat. Your ticket specifies the stadium and date along with the location of the seat.

Another way to explain this array indexing business is to consider an array of desks in a classroom, more commonly known as a seating chart. Imagine that it is the first school day. Some teacher believes in assigned seats and also wants to be organized. Each student is given an assigned seat number. The teacher's seating chart, for *Room J116* below, is one example.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [16]  Ingrid | [17]  Darlene | [18]  Gene | [19]  Sean | [20]  Stephanie |
| [11]  Holly | [12]  Blake | [13]  Michelle | [14]  Remy | [15]  Haley |
| [06]  Diana | [07]  Jessica | [08]  David | [09]  Anthony | [10]  Alec |
| [01]  Isolde | [02]  John | [03]  Greg | [04]  Maria | [05]  Heidi |

The seating chart demonstrates a very important array feature. Do not confuse the array index, with the contents at the array location, specified by the index. In other words, **J116[08]** is the desk for **David**. Room **J116**, seat number **[08]** is not equal to **David**. It is the location where David is supposed to sit.

As you look at the examples in this chapter, and as you write your own programs with an array data structure, be aware of the difference between *index* and *content*.

**10.3 1D Array Declaration and Access**

Consider program **Java1001.java**, in figure 10.1. This program has ten integer declarations, assigns ten integer values to the variables and then displays the values of each one of the ten variables. It may seem that working with ten variables, as shown here, is not that bad. But is there a better way?

**Figure 10.1**

|  |
| --- |
| // Java1001.java  // This program declares 10 different <int> variables.  // Each variable is assigned a value and each variable value is displayed.  // This approach is very inefficient for a large number of variables.  public class Java1001  {  public static void main(String args[])  {  System.out.println("Java1001\n");  int number0 = 100;  int number1 = 101;  int number2 = 102;  int number3 = 103;  int number4 = 104;  int number5 = 105;  int number6 = 106;  int number7 = 107;  int number8 = 108;  int number9 = 109;    System.out.print(number0 + " ");  System.out.print(number1 + " ");  System.out.print(number2 + " ");  System.out.print(number3 + " ");  System.out.print(number4 + " ");  System.out.print(number5 + " ");  System.out.print(number6 + " ");  System.out.print(number7 + " ");  System.out.print(number8 + " ");  System.out.print(number9 + " ");    System.out.println();  }  } |

|  |
| --- |
| **Java1001.java Output**  Java1001  100 101 102 103 104 105 106 107 108 109 |

Program **Java1002.java**, in figure 10.2, does not appear very different from the previous program. The similarity with the previous program is by the fact that ten integer values are assigned to ten variables. Then the program continues and displays each one of the ten values. However, there is something odd about these ten variables. There is some strange looking operator in use with a set of square brackets. Additionally, it seems that every one of the ten variables is called **list**. You are actually looking at the declaration of an integer array. Specifically, this is a declaration for a single variable, called **list**, which can store ten integer variables. Many streets have an array of homes. The street has a single name, like **Main Street** or **King Road**. A street can be considered an array of homes. Since there is a single street name, it becomes necessary to give a label to each home that identifies the home. Each home has a street number. In the case of the **list** array, each element of the array has an index, which is placed between brackets.

**Figure 10.2**

|  |
| --- |
| // Java1002.java  // This program declares an array of 10 <int> elements.  // Each array element value is individually assigned and displayed.  // There does not appear any real benefit from the previous program example.  public class Java1002  {  public static void main(String args[])  {  System.out.println("Java1002\n");  int list[]; // declares the array object identifier  list = new int[10]; // allocates memory for 10 array elements  list[0] = 100;  list[1] = 101;  list[2] = 102;  list[3] = 103;  list[4] = 104;  list[5] = 105;  list[6] = 106;  list[7] = 107;  list[8] = 108;  list[9] = 109;  System.out.print(list[0] + " ");  System.out.print(list[1] + " ");  System.out.print(list[2] + " ");  System.out.print(list[3] + " ");  System.out.print(list[4] + " ");  System.out.print(list[5] + " ");  System.out.print(list[6] + " ");  System.out.print(list[7] + " ");  System.out.print(list[8] + " ");  System.out.print(list[9] + " ");  System.out.println();  }  } |

|  |
| --- |
| **Java1002.java Output**  Java1002  100 101 102 103 104 105 106 107 108 109 |

Figure 10.3 displays a segment of the previous program. Let us examine each one of the program statements.

**Figure 10.3**

|  |
| --- |
| int list[]; // line 1  list = new int[10]; // line 2  list[0] = 100; // line 3  list[1] = 101; // line 4  list[2] = 102; // line 5 |

Line 1 declares variable **list** to be an array of **int** values.

Line 2 allocates space with the **new** operator for ten **int** values in the **list** array.

Line 3 assigns value **100** to the first **list** space. Do not get confused, because access to array elements is done by using **index [0]** for the first element. This also means that the index of the last element of an array is always one less than the number of elements in the array.

Program **Java1002.java**, which used an array variable, did not seem to provide much of an improvement to program **Java1001.java**, which used ten variables. Both programs appeared functional and they were both about the same length. Now look at program **Java1003.java**, in figure 10.4. You see far fewer statements, yet it generates the same result as the two previous program examples.

**Figure 10.4**

|  |
| --- |
| // Java1003.java  // The previous program - with separate statements for each array member assignment  // and display - is now replaced with two loops. The loop counter index is used  // to specify each array element in an efficient manner.  public class Java1003  {  public static void main(String args[])  {  System.out.println("Java1003\n");  int list[];  list = new int[10];  for (int index = 0; index <=9; index++)  list[index] = index + 100;  for (int index = 0; index <=9; index++)  System.out.print(list[index] + " ");    System.out.println();  }  } |

**Figure 10.4 Continued**

|  |
| --- |
| **Java1003.java Output**  Java1003  100 101 102 103 104 105 106 107 108 109 |

Figure 10.5 isolates how cleverly an array uses a loop structure. In this example the loop repeats ten times. The loop counter, called **index**, starts at **0** and ends at **9**, which is the range of the **list** index values. Previously, you saw statements, like **list[4] = 300;** but now in place of a fixed integer, an integer variable (**index**) is used to assign ten values to ten **list** locations.

**Figure 10.5**

|  |
| --- |
| **for (int index = 0; index <=9; index++)**  **list[index] = index + 100;** |

Program **Java1004.java**, in figure 10.6, repeats the looping access shown in the previous program. This time the array declaration is done in a single statement rather than the two statements used previously.

**Figure 10.6**

|  |
| --- |
| // Java1004.java  // This program is the same list array and the same list values as the previous program.  // This time note that the array declaration and object construction is accomplished with  // a single statement.  public class Java1004  {  public static void main(String args[])  {  System.out.println("Java1004\n");  int list[] = new int[10];  for (int index = 0; index <=9; index++)  list[index] = index + 100;  for (int index = 0; index <=9; index++)  System.out.print(list[index] + " ");  }  } |

|  |
| --- |
| **Java1004.java Output**  Java1004  100 101 102 103 104 105 106 107 108 109 |

Figure 10.7 pulls the array declarations from two previous programs. The first example uses two statements. The second example combines the two statements into one. The one-statement approach is more commonly used.

**Figure 10.7**

|  |
| --- |
| **Declaring an array with two program statements**  **int list[ ];**  **list = new int[10];** |
| **Declaring an array with one program statement**  **int list[ ] = new int[10];** |

It is not necessary to declare the size of an array if an ***initializer list*** is used. Program **Java1005.java**, in figure 10.8, shows a set of integer values placed between braces. This type of syntax tells the computer that **list** is an integer array with ten elements and it simultaneously assigns the ten provided values.

**Figure 10.8**

|  |
| --- |
| // Java1005.java  // This program demonstrates how to initialize array elements.  // The <new> operator is not necessary in this case.  public class Java1005  {  public static void main(String args[])  {  System.out.println("Java1005\n");  int list[] = {100,101,102,103,104,105,106,107};  for (int k = 0; k <= 7; k++)  System.out.println("list[" + k + "] = " + list[k]);  System.out.println();  }  } |

|  |
| --- |
| **Java1005.java Output**  Java1005  list[0] = 100  list[1] = 101  list[2] = 102  list[3] = 103  list[4] = 104  list[0] = 105  list[0] = 106  list[0] = 107 |

What you have seen so far is an integer array called **list**. Arrays are not limited to storing integer values. Figure 10.9 repeats **list** as an **int** array, continues with **names** as a **char** array, and finishes with **grades** as a **double** array.

**Figure 10.9**

|  |
| --- |
| int list[ ]; // declares the array **list** identifier  list = new int[10]; // allocates memory for 10  **int** variables  char name[ ]; // declares the **name** array identifier  name = new char[25]; // allocates memory for 25 **char** variables  double grades[ ]; // declares the **grades** array identifier  grades = new double[50]; // allocates memory for 50 **double** variables |

Program **Java1006.java**, in figure 10.10, demonstrates that it is no problem to create arrays with data values besides integers. The same syntax is used for initializer lists that contain characters and strings. It is important to realize that character values require singles quotes and string values require double quotes.

**Figure 10.10**

|  |
| --- |
| // Java1006.java  // This program demonstrates a <char> array and a <String> array.  // Both arrays use an initializer list.  public class Java1006  {  public static void main(String args[])  {  System.out.println("Java1006\n");    char list1[] = {'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'};  for (int k = 0; k < 26; k++)  System.out.print(list1[k]);  System.out.println("\n");    String list2[] = {"John","Greg","Maria","Heidi","Diana","David"};  for (int k = 0; k < 6; k++)  System.out.print(list2[k]);  System.out.println();  }  } |

|  |
| --- |
| **Java1006.java Output**  Java1006  ABCDEFGHIJKLMNOPQRSTUVWXYZ  John Greg Maria Heidi Diana David |

There is a problem with this program in that it is not flexible. Look specifically at **list2**. Suppose you were to add some names to the end of the list. When you execute the program, those names will not show up because the statement **for(int k = 0; k < 6; k++)** will only display the first 6 names in the array. What if you remove names from the array? The program will then try to display the first 6 names of an array with less than 6 names. The program will simply crash.

Program **Java1007.java**, in figure 10.11, fixes this problem by using the **length** keyword. Java Static Arrays provide a convenient way to know how large a particular array is. It is possible to keep track of the size of an array with a user-defined variable, but you can use **length** at any time to retrieve the array size.

Try adding names to or removing names from the list as you did in the previous program. You will notice this program is more flexible. Since **length** is used instead of a hard coded number **6**. The **for** loop is controlled by the actual number of elements in the array, however many that may be.

The **length** keyword is somewhat of a strange creature. You may be tempted to use syntax like **list.length()**, because the *dot notation* does give the impression that **length** is a method of the **list** object; however, **length** is NOT a method. **length** is actually a *field* similar to the **PI** field of the **Math** class.

**Figure 10.11**

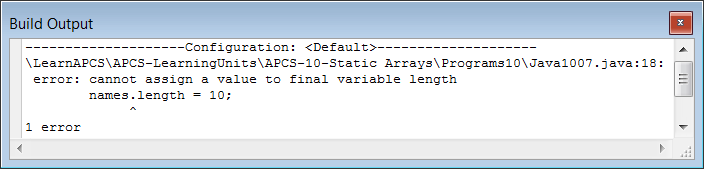
|  |
| --- |
| // Java1007.java  // This program introduces the length field to determine the  // number of elements in the array. Remove the comments from line 18  // to observe what happens when the length field is altered.  public class Java1007  {  public static void main(String args[])  {  System.out.println("Java1007\n");  String names[] = {"Joe","Tom","Sue","Meg"};  **int n = names.length;** // data field access; not a method call  System.out.println("There are " + n + " array elements.");  for(int k = 0; k < n; k++)  System.out.println("names[" + k + "] = " + names[k]);  **// names.length = 10;**  }  } |

|  |
| --- |
| **Java1007.java Output**  Java1007  There are 4 array elements.  names[0] = Joe  names[1] = Tom  names[2] = Sue  names[3] = Meg |

Near the end of program **Java1007.java**, you may have noticed this peculiar statement:

**// names.length = 10;**

This statement is currently commented out of the program. If the comment symbols are remove the program will not compile as you can see below:



Note the error message refer to **length** as a *final variable*. This means changing **length** is like trying to change the value of **Math.PI** or any other constant. It simply is not allowed. Here is a simple fact. What makes static arrays *static* is that you cannot change their size. If you create a static array with 4 names, it will always have 4 names. In the next chapter you will learn about *dynamic arrays*. The size of a *dynamic array* can be changed. For now we will stick with *static arrays*.

**10.4 Random Arrays**

Hard coding array elements during array definitions can be tedious. This is especially tedious if there are many arrays values. Entering array elements during program execution also becomes quite annoying when there are many numbers to be added. Frequently, in the process of learning computer science it is important to process a large quantity of data, and such data in many cases does not need to be any specific values. In other words, random data is just fine. The only issue left is the nature of the random data. Are integers, real numbers, characters or strings required? In this brief section you will learn how to assign a variety of data types randomly to an array.

In an earlier chapter you learned how to generate random integers with the **Random** class and the **nextInt** method. You also learned how to construct **Random** objects with a *seed* passed to the constructor. The seed insures that every program execution produces an identical set of numbers. If you are rusty with controlling random integers, go back to Chapter VI and see how the **Random** class and the **nextInt** method can be used to generate any desired range of random integers.

Program **Java1008.java**, in figure 10.12, demonstrates how to generate random numbers in a specified range, which is [100..999] in this case, and assigns these numbers to a sequence of integer array elements.

**Figure 10.12**

|  |
| --- |
| // Java1008.java  // This program fills an <int> array with a random set of numbers.  import java.util.Random; // necessary to use the <Random> class  public class Java1008  {  public static void main(String args[])  {  System.out.println("Java1008\n");  int list[] = new int[20];  Random random = new Random(12345);  for (int k = 0; k < 20; k++)  list[k] = random.nextInt(900) + 100;  for (int k = 0; k < 20; k++)  System.out.println("list[" + k + "] = " + list[k]);  System.out.println();  }  } |

|  |
| --- |
| **Java1009.java Output**  Java1008  List[0] = 851  List[1] = 680  List[2] = 241  List[3] = 928  List[4] = 458  List[5] = 284  List[6] = 575  List[7] = 802  List[8] = 701  List[9] = 889  List[10] = 717  List[11] = 142  List[12] = 890  List[12] = 206  List[14] = 312  List[15] = 584  List[16] = 687  List[17] = 803  List[18] = 432  List[19] = 775 |

Several years ago I saw an advertisement for a solder action figure. This was not a cheap piece of plastic found in a bucket of 100 soldiers. This 12 inch action figure was fairly sophisticated. It had a built-in voice synthesizer so that it could speak. The commercial claimed the action figure could speak over half a million different commands. You might be thinking that somewhere in the world, there was some poor guy that had to type 500,000+ different sentences for this action figure. That is not how it worked at all. In the previous program you saw an array that was filled with random data. What if the data is not random at all, but instead the index is random. This would allow you to access the information randomly. This is what is done in program **Java1009.java**, shown in figure 10.13. This program will not generate the same 500,000+ different commands, but it will generate 2400+. Here is how it works. There are 4 different arrays. One has 7 *ranks*; one has 7 *names*; one has 7 *actions* and the last one has 7 *locations*. 4 random indexes are chosen, one for each array. This allows me to create a sentence with a random **rank**, followed by a random **name**, followed by a random **action** followed by a random **location**. With 7 items in each array there are 7 \* 7 \* 7 \* 7 or 2401 different sentences possible. The action figure from several years ago did something very similar, but it had about 27 items in each array. This means 27 \* 27 \* 27 \* 27 = 531,441 different sentences possible.

Note, in this example the arrays store strings. Arrays can also store integers, real numbers, characters, boolean values and other things as well. Regardless of what the array stores, the index is always an **int** value.

**Figure 10.13**

|  |
| --- |
| // Java1009.java  // This program will display 15 random sentences.  // With 7 different ranks, 7 different people, 7 different actions and 7 different locations,  // there are more than 2400 different sentences possible.  import java.util.Random; // necessary to use the <Random> class  public class Java1009  {  public static void main(String args[])  {  System.out.println("Java1009\n");  Random random = new Random();  String rank[] = {"Private", "Corporal", "Sargent", "Lieutenant", "Captain", "Major", "General"};  String person[] = {"Smith", "Gonzales", "Brown", "Jackson", "Powers", "Jones", "Nguyen"};  String action[] = {"drive the tank", "drive the jeep", "take the troops", "bring all supplies",  "escort the visitor", "prepare to relocate", "bring the Admiral"};  String location[] = {"over the next hill", "to the top of the mountain", "outside the barracks",  "30 miles into the dessert", "to the middle of the forest",  "to my present location", "to anywhere but here"};  for (int j = 1; j <= 15; j++)  {  int randomRank = random.nextInt(rank.length);  int randomPerson = random.nextInt(person.length);  int randomAction = random.nextInt(action.length);  int randomLocation = random.nextInt(location.length);  String sentence = rank[randomRank] + " " + person[randomPerson] + " " +  action[randomAction] + " " + location[randomLocation] + ".";  System.out.println("\n" + sentence);  }  System.out.println();  }  } |

**Figure 10.13 Continued**

|  |
| --- |
| **Java1009.java Output**  Java1009  Major Gonzales take the troops to anywhere but here.  Major Powers bring the Admiral over the next hill.  Corporal Jones drive the jeep to the middle of the forest.  General Jones take the troops to my present location.  Major Smith prepare to relocate over the next hill.  General Nguyen drive the jeep to the middle of the forest.  Captain Brown escort the visitor to the middle of the forest.  Sargent Smith escort the visitor to anywhere but here.  General Powers drive the jeep 30 miles into the dessert.  General Jackson drive the jeep to my present location.  Sargent Brown bring the Admiral over the next hill.  Sargent Powers prepare to relocate over the next hill.  Major Gonzales prepare to relocate to anywhere but here.  Sargent Jackson bring the Admiral 30 miles into the dessert.  General Jones escort the visitor outside the barracks. |

**10.5 Accessing Array Elements with the <for..each> Loop Structure**

The **for** loop was introduced back in Chapter V, at least the type-of **for** loop that uses a *loop control variable*. Java version 5.0 introduced an *enhanced for* loop. This *new-and-improved* loop structure can manage access to array elements without using an index. A loop counter is not necessary, because the loop iterates until every element of the array has been accessed.

Program **Java1010.java**, in figure 10.14, compares the old **for**-loop with the new **for** loop. You will note that the statements of the enhanced **for** loop has simpler syntax. Inside the parentheses you need to use the name of the data structure, like **list** and a variable that is the same type as each array element.

**Figure 10.14**

|  |
| --- |
| // Java1010.java  // This program introduces the Java Version 5.0 enhanced <for..each> loop with an <int> array.  public class Java1010  {  public static void main(String args[])  {  System.out.println("Java1010\n");  int list[] = {11,22,33,44,55,66,77,88,99};    for (int k = 0; k < 9; k++) // Old <for> loop syntax  System.out.print(list[k] + " ");  System.out.println("\n\n");    for (int item: list) // New <for> loop syntax  System.out.print(item + " ");  System.out.println("\n\n");  }  } |

|  |
| --- |
| **Java1010.java Output**  Java1010  11 22 33 44 55 66 77 88 99  11 22 33 44 55 66 77 88 99 |

Program **Java1011.java**, in figure 10.15, uses the new **for** loop to display the elements of a **String** array. Note the use of the identifiers. The array data structure is called **names**, implying more than one value. The identifier of each array element is called **name**, implying a single value. It is a small detail, but it is part of good program design to use practical, self-documenting identifiers. With small programs, program readability is not such a big deal. As programs grow and start to become thousands of lines of program code, you will truly appreciate identifiers that clearly indicate the nature of the stored value.

**Figure 10.15**

|  |
| --- |
| // Java1011.java  // This program uses the Java Version 5.0 <for..each> loop with a <String> array.  public class Java1011  {  public static void main(String args[])  {  System.out.println("Java1011\n");  String names[] = {"Tom","Sue","Joe","Jan","Bob","Lee","Ann","Meg"};    for (int k = 0; k < 8; k++)  System.out.print(names[k] + " "); // Old <for> loop syntax  System.out.println("\n\n");    for (String name: names) // New <for> loop syntax  System.out.print(name + " ");  System.out.println("\n\n");  }  } |

|  |
| --- |
| **Java1011.java Output**  Java1011  Tom Sue Joe Jan Bob Lee Ann Meg  Tom Sue Joe Jan Bob Lee Ann Meg |

The new **for** loop can be used with many different data structures. Essentially, an object is required. Program **Java1010.java**, in figure 10.16, demonstrates an array of **String** values. This creates an object of objects. The **names** array is an object and each array element, which is a **String** value, is also an object. This means that the new **for** loop can use the general data type **Object**.

This generalization business will not seem quite so significant right now, but in the near future you will find that there are good reasons to make generalizations about your data types. Many text books, including this one, use a technique whereby the foundation for a future topic or concept is provided early. When the new topic is truly introduced in a serious manner, students already have some elementary insight. This approach helps to eliminate *new-topic-anxiety*. At least that is the aim. If anxiety is not relieved, well I am sorry. I did try.

**Figure 10.16**

|  |
| --- |
| // Java1012.java  // This program demonstrates a very generalized <for. .each> loop usage  // with the <Object> class.  public class Java1012  {  public static void main(String args[])  {  System.out.println("Java1012\n");  String names[] = {"Tom","Sue","Joe","Jan","Bob","Lee","Ann","Meg"};    for (int k = 0; k < 8; k++)  System.out.print(names[k] + " ");  System.out.println("\n\n");    for (**Object** obj: names)  System.out.print(obj + " ");  System.out.println("\n\n");  }  } |

|  |
| --- |
| **Java1012.java Output**  Java1012  Tom Sue Joe Jan Bob Lee Ann Meg  Tom Sue Joe Jan Bob Lee Ann Meg |

|  |
| --- |
| **Enhanced for Loop** |
| **The enhanced for loop is called the "for..each" loop.**  **This loop structure is available with Java 5.0.**  **The new loop structure does not replace the older for loop, because it is not possible to access specific array elements.**  **int numbers[ ] = {1,2,3,4,5,6,7,8,9};**  **for (int number: numbers)**  **System.out.print(number + " ");** |

**10.6 Introduction to Static 2D Arrays**

All the array examples reviewed so far in this chapter have had something in common. They have all been examples of one-dimensional (1D) arrays. The term *one-dimensional* may have been used a few times without much emphasis on the concept. It is easier to discuss the dimension of an array when there is the opportunity to make comparisons. In real life you are surrounded by examples of both one-dimensional and two-dimensional arrays. A street is a one-dimensional array of homes. Reserved seats at a football stadium and seats in an airplane represent a two-dimensional array. Take a look at the seating in figure 10.17. An array of twenty desks is shown. These twenty desks can be viewed as a linear sequence of seats, which are numbered from 1 to 20. We could make a closer analogy to Java arrays and number the seats from 0 to 19. However, we can select to represent the seats in a different manner that realistically resembles the physical desk layout better using a two-dimensional array.

**Figure 10.17**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [16]  Ingrid | [17]  Darlene | [18]  Gene | [19]  Sean | [20]  Stephanie |
| [11]  Holly | [12]  Blake | [13]  Michelle | [14]  Remy | [15]  Haley |
| [06]  Diana | [07]  Jessica | [08]  David | [09]  Anthony | [10]  Alec |
| [01]  Isolde | [02]  John | [03]  Greg | [04]  Maria | [05]  Heidi |

The classroom has twenty desks and each desk has a label or index. Labels start at [01] and end at [20]. This is all simple enough, but the seating chart resembles a matrix or two-dimensional array more closely. The class has rows and columns. In fact there are four rows of desks and five columns of desks.

It is possible to designate each student with double labels, one for row and a second one for column. On the second seating chart, in figure 10.18, you will see the same classroom and the same desk arrangement, but this time there are two labels for each seat. The first index, by convention, indicates the row location and the second index labels the column.

**Figure 10.18**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [04][01]  Ingrid | [04][02]  Darlene | [04][03]  Gene | [04][04]  Sean | [04][05]  Stephanie |
| [03][01]  Holly | [03][02]  Blake | [03][03]  Michelle | [03][04]  Remy | [03][05]  Haley |
| [02][01]  Diana | [02][02]  Jessica | [02][03]  David | [02][04]  Anthony | [02][05]  Alec |
| [01][01]  Isolde | [01][02]  John | [01][03]  Greg | [01][04]  Maria | [01][05]  Heidi |

The example of the seating chart actually goes a little against another convention. Two-dimensional arrays are frequently shown as a matrix of values, and the first row in the matrix is normally shown at the top of the matrix. This seating chart has intentionally been displayed from the point of view of the instructor. Using two values to indicate a location is very common. A Football ticket indicates something like *Row K, Seat 34*, and an airline boarding pass may say *Row 29, Seat D*. In the world of mathematics, science and business, a matrix of numbers is used for many types of calculations.

Declaring a two-dimensional array holds few surprises. Everything you have learned about one-dimensional arrays in the areas of declarations and definitions is identical. You need an array identifier along with a data type for each element of the array. You also need the **new** operator to define the size of the array.

There is only one small difference you will see in program **Java1013.java**, shown in figure 10.19, and that is the need to provide two sets of brackets. The first set of brackets dimensions the number of rows and the second set of brackets dimensions the number of columns in the array. In the first 2D array example every array element is accessed individually. This works, and it is manageable for a small array of two rows by three columns.

**Figure 10.19**

|  |
| --- |
| // Java1013.java  // This program introduces 2D Java static arrays.  // For 2D arrays two sets of index operators are needed.  // The first set of index brackets stores the rows value.  // The second set of index operators stores the cols value.  // The <twoD> array in this program is a 2 X 3 array.    public class Java1013  {    public static void main(String args[])  {  System.out.println("\nJava1013.java\n");    int twoD[][]; // declaration of two-dimensional integer array  twoD = new int[2][3]; // new 2D array is constructed with 2 rows and 3 columns  twoD[0][0] = 1;  twoD[0][1] = 2;  twoD[0][2] = 3;  twoD[1][0] = 4;  twoD[1][1] = 5;  twoD[1][2] = 6;    System.out.print(twoD[0][0] + " ");  System.out.print(twoD[0][1] + " ");  System.out.print(twoD[0][2] + " ");  System.out.println();  System.out.print(twoD[1][0] + " ");  System.out.print(twoD[1][1] + " ");  System.out.print(twoD[1][2] + " ");  System.out.println();  }  } |

|  |
| --- |
| **Java1013.java Output**  Java1013.java  1 2 3  4 5 6 |

Program **Java1014.java**, in figure 10.20, is the typical approach used with 2D arrays. Like the earlier 1D arrays, loop control structures are used. In the case of 2D arrays two separate loops are needed. An outer loop controls the rows and the inner loop controls the columns of the 2D array.

**Figure 10.20**

|  |
| --- |
| // Java1014.java  // This program stores and displays the same values in a 2 X 3 array  // as the previous program.  // But this time a set of nested loops is used with 2D arrays to assign  // and display individual values.  // Additionally, the declaration of the 2D array is done in one statement.  public class Java1014  {  public static void main(String args[])  {  System.out.println("\nJava1014.java\n");  int twoD[][] = new int[2][3]; // 2D array declaration in one statement.    int count = 1;  for (int row = 0; row < 2; row++)  {  for (int col = 0; col < 3; col++)  {  twoD[row][col] = count;  count++;  }  }    for (int row = 0; row < 2; row++)  {  for (int col = 0; col < 3; col++)  {  System.out.print(twoD[row][col] + " ");  }  System.out.println();  }  System.out.println();  }  } |

|  |
| --- |
| **Java1014.java Output**  Java1014.java  1 2 3  4 5 6 |

The handy initializer list, shown with the 1D arrays, also is available for the 2D arrays. It helps to realize that technically speaking there are no 2D arrays. A 2D array is really a 1D array where every array element is a 1D array. In other words, a 2D array is *an array of arrays.* Program **Java1015.java**, in figure 10.21, uses an initializer list. Observe the comma placement and you see the *array of arrays* concept visualized. A second approach, which visually displays the 2D matrix, is also shown in comments.

**Figure 10.21**

|  |
| --- |
| // Java1015.java  // This program demonstrates how to use an initializer list  // with a 2D array to assign values.  // Commented lines 16 and 17 show a second style of using initializer  // lists with 2D arrays that display the matrix appearance.  public class Java1015  {  public static void main(String args[])  {  System.out.println("\nJava1015.java\n");  int twoD[][] = { {1,2,3}, {4,5,6} };  // int twoD[][] = { {1,2,3},  // {4,5,6} };  for (int row = 0; row < 2; row++)  {  for (int col = 0; col < 3; col++)  {  System.out.print(twoD[row][col] + " ");  }  System.out.println();  }  System.out.println();  }  } |

|  |
| --- |
| **Java1015.java Output**  Java1015.java  1 2 3  4 5 6 |

Program **Java1016.java**, in figure 10.22, may seem similar to earlier programs, but it displays a very common error that causes array programs to crash during program execution.

The problem with this program is that the number of rows is confused with the number of columns. The matrix has 7 rows and 5 columns. When the program attempts to display 5 rows with 7 columns, the program crashes with a runtime exception. More information about Java exceptions will be discussed later in a later chapter. The program crashes because an index value is too big.

**Figure 10.22**

|  |
| --- |
| // Java1016.java  // This program demonstrates what happens when rows and columns are confused.  // A matrix of 7 rows and 5 columns is created.  // The program attempts to display 5 rows and 7 columns.  // The program will compile and execute, but then it will crash mid-display.    public class Java1016  {  public static void main(String args[])  {  System.out.println("\nJava1016.java\n");  int k = 1;  int matrix[][] = new int[7][5]; // 7 rows and 5 columns  for (int r = 0; r < 7; r++)  for (int c = 0; c < 5; c++)  {  matrix[r][c] = k;  k++;  }  System.out.println();    for (int r = 0; r < 5; r++) // should be 7  {  for (int c = 0; c < 7; c++) // should be 5  System.out.print(matrix[r][c] + " ");  System.out.println();  }  System.out.println();  }  } |

|  |
| --- |
| **Java1016.java Output**  Java1016.java  1 2 3 4 5 Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5  at Java1016.main(Java1016.java:26) |

2D arrays normally involve matrices and matrices are usually displayed in a *row x col* configuration. This does involve lining up the columns correctly for a pleasing output display. Program **Java1017.java**, in figure 10.23, is logically correct and the output is also correct, but the matrix does not look very neat when the numerical values have a different numbers of digits.

**Figure 10.23**

|  |
| --- |
| // Java1017.java  // This program allows the user to specify the number of rows and columns.  // Note that the output will not line up nicely as it combines single,  // double and triple digit numbers.  import java.util.Scanner; // necessary to use the <Scanner> class    public class Java1017  {  public static void main(String args[])  {  System.out.println("\nJava1017.java\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter the number of rows --> ");  int numRows = input.nextInt();  System.out.print("Enter the number of columns --> ");  int numCols = input.nextInt();  int k = 1;  int matrix[][] = new int[numRows][numCols];  for (int r = 0; r < numRows; r++)  for (int c = 0; c < numCols; c++)  {  matrix[r][c] = k;  k++;  }  System.out.println();  for (int r = 0; r < numRows; r++)  {  for (int c = 0; c < numCols; c++)  System.out.print(matrix[r][c] + " ");  System.out.println();  }  System.out.println();  }  } |

|  |
| --- |
| **Java1017.java Output**  Java1017.java  Enter the number of rows --> 3  Enter the number of columns --> 4  1 2 3 4  5 6 7 8  9 10 11 12 |

**10.7 Controlling 2D Array Output**

When two-dimensional arrays store numerical data, it is customary to display the output of such arrays in the **row x col** format shown in the previous program. That program also shows that numbers of different sizes can make the resulting output undesirable.

One solution is to create a **DecimalFormat** object, which can then control the output of numerical data. Program **Java1018.java**, in figure 10.24 uses the **DecimalFormat** class to make all numbers display as 3 digit numbers. Now all the numbers, whether they are ones, ten's, or hundred's line up as desired.

**Figure 10.24**

|  |
| --- |
| // Java1018.java  // This program demonstrates using <DecimalFormat> with 2D arrays.  // By using <DecimalFormat> we can make output line up properly.  **import java.text.DecimalFormat; // necessary to use the <DecimalFormat> class**  import java.util.Scanner; // necessary to use the <Scanner> class    public class Java1018  {  public static void main(String args[])  {  System.out.println("\nJava1018.java\n");  **DecimalFormat threeDigits = new DecimalFormat("000");**  Scanner input = new Scanner(System.in);  System.out.print("Enter the number of rows --> ");  int numRows = input.nextInt();  System.out.print("Enter the number of columns --> ");  int numCols = input.nextInt();  int k = 1;  int matrix[][] = new int[numRows][numCols];  for (int r = 0; r < numRows; r++)  for (int c = 0; c < numCols; c++)  {  matrix[r][c] = k;  k++;  }  System.out.println();  for (int r = 0; r < numRows; r++)  {  for (int c = 0; c < numCols; c++)  **System.out.print(threeDigits.format(matrix[r][c]) + " ");**  System.out.println();  }  System.out.println();  }  } |

**Figure 10.24 Continued**

|  |
| --- |
| **Java1018.java Output**  Java1018.java  Enter the number of rows --> 3  Enter the number of columns --> 4  001 002 003 004  005 006 007 008  009 010 011 012 |

There are computer scientists who feel that we really do not have multi-dimensional arrays. We only have one-dimensional arrays and some of these arrays are manipulated in a manner that makes them seem to be two-dimensional.

This may seem odd, but there is compelling evidence for this approach. Ask yourself, what can you store in a one-dimensional array? Well there are numbers, characters, strings that can be stored and yes you can also store an array in each element of an existing array. This becomes *an array of arrays*.

In real life that actually is also what happens. Suppose you go to a concert with a reserved seat ticket. Your ticket states *Row H, Seat 17*. Now here is a question: Do you go straight to your seat? I would rather doubt that. Your first mission is to find ***Row H***. You walk down the aisle and there is ***H***. At this location you are now looking at a one-dimensional array of seats. This one-dimensional array is called ***H***. The concert venue is a two-dimensional array of seats, but it can also be called a one-dimensional array of rows, which then in turn is another one-dimensional array of seats.

You may be puzzled, because you have just started a section on 2D array output and this *array of arrays* seems to go off to some weird tangent. There is a reason for this explanation. The next program example shows how to use the special <for..each> loop with a 2D array. The format of the two-dimensional <for..each> loop is based on the *array of arrays* logic.

Program **Java1019.java**, in figure 10.25, is the same as the previous program, except for the fact that the output display is now handled by a <for..each> loop. Examine the program and its output briefly and then continue and concentrate on the isolated <for..each> loop portion in figure 10.26.

**Figure 10.25**

|  |
| --- |
| // Java1019.java  // This program uses the <for..each> loop to display array data.  // It also helps to illustrate that a 2D array is an array of arrays.  import java.text.DecimalFormat; // necessary to use the <DecimalFormat> class  import java.util.Scanner; // necessary to use the <Scanner> class  public class Java1019  {  public static void main(String args[])  {  System.out.println("\nJava1018.java\n");  DecimalFormat threeDigits = new DecimalFormat("000");  Scanner input = new Scanner(System.in);  System.out.print("Enter the number of rows --> ");  int numRows = input.nextInt();  System.out.print("Enter the number of columns --> ");  int numCols = input.nextInt();  System.out.println("\n");  int k = 1;  int matrix[][] = new int[numRows][numCols];  for (int r = 0; r < numRows; r++)  for (int c = 0; c < numCols; c++)  {  matrix[r][c] = k;  k++;  }  System.out.println();  for (int[] row: matrix)  {  for (int number: row)  System.out.print(threeDigits.format(number) + " ");  System.out.println();  }  System.out.println();  }  } |

|  |
| --- |
| **Java1019.java Output**  Java1019.java  Enter the number of rows --> 3  Enter the number of columns --> 4  001 002 003 004  005 006 007 008  009 010 011 012 |

**Figure 10.26**

|  |
| --- |
| **for (int[ ] row: matrix)**  **{**  **for (int number: row)**  **System.out.print(threeDigits.format(number) + " ");**  **System.out.println();**  **}** |

Does the nested <for..each> loop, in figure 10.26, make sense? The outer loop - in a concise, Java statement - says the following in English ...

***For each array element in matrix, which is an int array called row, do the following ...***

Well inside the outer <for..each> loop is an inner <for..each> loop, which continues with its own Java logic and once again we can say in English ...

***For each array element in row, which is an int called number, display the number.***

Now compare the <for..each> loop structure in figure 10.26 to the <for> loop structure in figure 10.27, which uses the original <for> loop structure.

**Figure 10.27**

|  |
| --- |
| **for (int r = 0; r < numRows; r++)**  **{**  **for (int c = 0; c < numCols; c++)**  **System.out.print(threeDigits.format(matrix[r][c]) + " ");**  **System.out.println();**  **}** |

|  |
| --- |
| **for..each Loop Structure Limitation** |
| The **for..each** loop structure is *read only* for any type of data structure, which include the one-dimensional array and the  two-dimensional array. |

**10.8 Storing Objects in a Static Array**

It is possible that you have the impression that Java static arrays store only primitive data types. Such a conclusion would be logical as all program examples have done just that. In reality, Java static arrays store all data types, including objects. Program **Java1020.java**, in figure 10.28, shows such an example. In this program there is a **Student** class, which stores the **name** and **age** of a student. A loop is used to repeat the keyboard input for the data and then with each iteration the entered data is used to instantiate a new **Student** object ready to be stored as a member of the **students** array.

**Figure 10.28**

|  |
| --- |
| // Java1020.java  // All the previous static arrays examples showed the use of a static  // array to stored primitive, simple data type values.  // This program example demonstrates that a static array can store  // object values as well. In this case a static array is constructed,  // which stores <Student> objects.  import java.util.Scanner;      public class Java1020  {    public static void main(String args[])  {  System.out.println("\nJava1020.java\n");  Scanner stringInput = new Scanner(System.in);  Scanner intInput = new Scanner(System.in);  System.out.print("Enter the number of students ==> ");  int numStudents = intInput.nextInt();    Student students[] = new Student[numStudents];  for (int index = 0; index < numStudents; index++)  {  System.out.print("Enter student's name ==> ");  String name = stringInput.nextLine();  System.out.print("Enter student's age ==> ");  int age = intInput.nextInt();  students[index] = new Student(name,age);  }    System.out.println();  for (int index = 0; index < numStudents; index++)  {  students[index].showData();  }  }    }  class Student  {  private String name;  private int age;    public Student(String n, int a)  {  name = n;  age = a;  }    public void showData()  {  System.out.println("Name: " + name);  System.out.println("Age: " + age);  System.out.println();  }  } |

**Figure 10.28 Continued**

|  |
| --- |
| **Java1020.java Output**  Java1020.java  Enter the number of students ==> 2  Enter student's name ==> George  Enter student's age ==> 23  Enter student's name ==> Rasha  Enter student's age ==> 19  Name: George  Age: 23  Name: Rasha  Age: 19 |

**10.9 2D Arrays and Length**

The **length** attribute or field was introduced earlier to determine the number of elements in a static array. At that time only one-dimensional arrays had been shown and a natural question is how size is handled with two-dimensional arrays? Consider program **Java1021.java**, in Figure 10.29.

**Figure 10.29**

|  |
| --- |
| // Java1021.java  // This program creates a 3 X 3 2D array and uses a method  // to display the array elements.  // The length field is used for both row and column length.  public class Java1021  {  public static void main(String args[])  {  System.out.println("\nJava1021.java\n");  int[][] mat = { {1,2,3},  {4,5,6},  {7,8,9} };  displayMatrix(mat);  }  public static void displayMatrix(int[][] m)  {  for (int r = 0; r < m.length; r++)  {  for (int c = 0; c < m.length; c++)  System.out.print(m[r][c] + " ");  System.out.println();  }  }  } |

|  |
| --- |
| **Java1021.java Output**  Java1021.java  1 2 3  4 5 6  7 8 9 |

This two-dimensional array seems to work quite nicely with the aid of **length**. There is a concern though, because a square matrix with three rows and three columns is used. The same **length** field is used for row size and column size. How well will this approach work if the matrix is not square?

Program **Java1022.java**, in Figure 10.30, has a 2 X 4 two-dimensional array. The output shows a 2 X 2 matrix. It appears that **length**, as it is used in the last two programs, is the value of the number of rows.

**Figure 10.30**

|  |
| --- |
| // Java1022.java  // The same <displayMatrix> method is used to display a  // 2 X 4 2D array. This time the method does not display correctly.  public class Java1022  {  public static void main(String args[])  {  System.out.println("\nJava1022.java\n");  int[][] mat = { {1,2,3,4},  {5,6,7,8} };  displayMatrix(mat);  }  public static void displayMatrix(int[][] m)  {  for (int r = 0; r < m.length; r++)  {  for (int c = 0; c < m.length; c++)  System.out.print(m[r][c] + " ");  System.out.println();  }  System.out.println();  }  } |

|  |
| --- |
| **Java1022.java Output**  Java1022.java  1 2  5 6 |

In the last section it was mentioned that a two-dimensional array is in fact a one-dimensional array or one-dimensional arrays. Program **Java1023.java**, in Figure 10.31, takes this new information and tries a different approach. **m.length** is used for the row size and then **m[0].length** is used for the column size. The program output does indicate that this approach works for an array that is not a square matrix. Keep in mind that **m[0]** is the first element of the **m** array, which has its own length.

**Figure 10.31**

|  |
| --- |
| // Java1023.java  // A very slight change with the column length results in the  // correct array display.  public class Java1023  {  public static void main(String args[])  {  System.out.println("\nJava1023.java\n");  int[][] mat = { {1,2,3,4},  {5,6,7,8} };  displayMatrix(mat);  }  public static void displayMatrix(int[][] m)  {  for (int r = 0; r < m.length; r++)  {  for (int c = 0; c < m[0].length; c++)  System.out.print(m[r][c] + " ");  System.out.println();  }  System.out.println();  }  } |

**Figure 10.31 Continued**

|  |
| --- |
| **Java1023.java Output**  Java1023.java  1 2 3 4  5 6 7 8 |

This **length** business may still be unclear. It might help to introduce an odd type of two-dimensional array that is not even part of the AP Computer Science curriculum. It is called a *ragged array*.

Program **Java1024.java**, in Figure 10.32, shows a two dimensional array with five rows, but the number of columns vary. The first row has one element; the second row has two elements, and so on. The **displayMatrix** method handles this varying length stuff without difficulty. **m.length** is used for the number of rows. The number of rows is a single value and does not change. On the other hand, each row has a different number of columns. **m[r].length** is used for the number of columns. Since **r** is a variable that stores the index value for each row, **m[r].length** will then store the number of columns for each different row.

**Figure 10.32**

|  |
| --- |
| // Java1024.java  // This program demonstrates how to construct an irregular two-dimensional array  // in the shape of a triangle, using a "ragged" array.  // It also shows how to use length for different column sizes.  public class Java1024  {  public static void main(String args[])  {  System.out.println("\nJava1024.java\n");  int[][] mat = { {1},  {1,2},  {1,2,3},  {1,2,3,4},  {1,2,3,4,5} };  displayMatrix(mat);  }    public static void displayMatrix(int[][] m)  {  for (int r = 0; r < m.length; r++)  {  for (int c = 0; c < m[r].length; c++)  System.out.print(m[r][c] + " ");  System.out.println();  }  System.out.println();  }  } |

**Figure 10.32 Continued**

|  |
| --- |
| **Java1024.java Output**  Java1024.java  1  1 2  1 2 3  1 2 3 4  1 2 3 4 5 |

|  |
| --- |
| **AP Examination Alert** |
| **Two-dimensional *ragged arrays*, as the shown in the example below are not tested on the APCS Examination.**  **int[][] mat = { {1},**  **{1,2},**  **{1,2,3},**  **{1,2,3,4},**  **{1,2,3,4,5}};** |

**2.10 Parameter Differences Between Simple Data Types and Arrays**

Parameter passing is not as easy as it may look. In this section we are going to explore what happens during parameter passing and how this impacts the variables in program segment where methods are called.

Perhaps the idea seems simplistic. The *actual* parameters of where the method is called provide values for the *formal* parameters of the method definition. Is there any change, any noticeable consequence to the actual parameters because of the method calling? We will examine three programs that use different data types to observe any consequences.

First, there is program **Java1025.java**, in Figure 10.33. In this program there are two **int** values passed to method **swap**. Method swap then uses a local variable **temp** to exchange the two integer values.

Execution of the program and the output window indicate that there is no swapping happening; at least not with the **p** and **q** variables. The **x** and **y** variables, inside the **swap** method, do exchange values.

**Figure 10.33**

|  |
| --- |
| // Java1025.java  // This program attempts to swap the p and q variable values,  // which does not work as expected.  public class Java1025  {  public static void main (String args[])  {  System.out.println("\nJava1025.java\n");  int p = 10;  int q = 20;  System.out.println("main, before swap " + p + " " + q);  swap(p,q);  System.out.println("main, after swap " + p + " " + q);  }  public static void swap(int x, int y)  {  System.out.println("swap start " + x + " " + y);  int temp = x;  x = y;  y = temp;  System.out.println("swap end " + x + " " + y);  }  } |

**Figure 10.33 Continued**

|  |
| --- |
| **Java1025.java Output**  Java1025.java  main, before swap 10 20  swap start 10 20  swap end 20 10  main, after swap 10 20 |

It may help to look at another program example. Program **Java1026.java**, in Figure 10.34, also exchanges the values of **x** and **y**. The output display shows the same consequence that was observed in the previous program. Swapping the **x**  and **y** values does not have any impact on variables **p** and **q**.

Without the **swap** method and parameter passing this may make sense, but the assignment statements **int x = p;** and **int y = q;** do behave in the same manner as the process of passing parameters.

A copy is made of the actual parameter (like **p)** and any change to the formal parameter (like **x**) will no longer have any impact on variable **p**.

**Figure 10.34**

|  |
| --- |
| // Java1026.java  // This program demonstrates the swapping process without a method.  // The swapping of variables x and y simulates the parameter passing concept.  // Note that changes made to variables x and y do not impact variable p and q.  public class Java1026  {  public static void main (String args[])  {  System.out.println("\nJava1026.java\n");  int p = 10;  int q = 20;  System.out.println("main, before swap " + p + " " + q);  /////////////////////////////////////////////////////////////////  int x = p;  int y = q;  System.out.println("swap start " + x + " " + y);  int temp = x;  x = y;  y = temp;  System.out.println("swap end " + x + " " + y);  ////////////////////////////////////////////////////////////////  System.out.println("main, after swap " + p + " " + q);  }  } |

**Figure 10.34 Continued**

|  |
| --- |
| **Java1026.java Output**  Java1026.java  main, before swap 10 20  swap start 10 20  swap end 20 10  main, after swap 10 20 |

It is quite possible that the two previous program examples leave you with a feeling of *so what?* Perhaps, the output of both programs makes perfect sense to you and you may wonder why this business is even mentioned in a chapter on static arrays?

Fair enough, so let us return to static arrays with program **Java1027.java**, in Figure 10.35, and also do some swapping with elements of a static array. As you look at the program example do not get confused. Parameters **3** and **6** are not values to be swapped, but they are the indexes of the two elements that need to be swapped, which are **44** and **77**, which do get exchanged. So why does **swap** work with the static arrays and not the primitive data types?

**Figure 10.35**

|  |
| --- |
| // Java1027.java  // This second example of using a <swap> method does swap the  // requested array elements.  public class Java1027  {  public static void main (String args[])  {  System.out.println("\nJava1027.java\n");  int[] list = {11,22,33,44,55,66,77,88,99};  System.out.println("Before swap " + list[3] + " " + list[6]);  swap(list,3,6);  System.out.println("After swap " + list[3] + " " + list[6]);  }  public static void swap(int[] x, int p, int q)  {  int temp = x[p];  x[p] = x[q];  x[q] = temp;  }  } |

**Figure 10.35 Continued**

|  |
| --- |
| **Java1027.java Output**  Java1027.java  Before swap 44 77  After swap 77 44 |

The answer is *shallow* and *deep* values. Simple or primitive data types, like **int**, **double**, **char** and **boolean** store their values in immediate or shallow memory locations allocated for the variables.

Data structures use a different system. The shallow memory location stores a reference to another set of *deeper* memory locations where a block of memory stores the actual integer values.

When parameters are passed, array actual array parameter **list** passes its shallow value (which is a memory reference) to formal array parameter **x.** Array **x** has a copy of the shallow value. We now have the situation that both **list** and **x** reference the exact same block of memory and when **x** starts to swap values, these are *deep* memory values, which are the same *deep* memory values as in **list**.

**10.11 Summary**

This chapter introduced the concept of a data structure. It is important to realize that a data structure is more than some means to store data. The manner in which the data is accessed is an integral part of a data structure.

The most common data structure in computer programming is the array. An array is a data structure with a fixed number of elements of the same type. Every element of the array can be accessed directly.

It is convenient to enter large sets of array values randomly. This can be done with a loop structure and the **random** method of the **Math** class. The **nextInt** method of the **Random** class can also be used. It is also possible to generate random string values by using a random index of a string array to specify some unknown string value.

Arrays can be declared for one, two or more dimensions. Each dimension requires its own index and access loop. One-dimensional arrays are also called vectors and two-dimensional arrays are called matrices.

Access to array elements can be done with index access using the older **for** loop. It is also possible with the enhanced Java 5.0 **for..each** loop, which does not require an index value. The **for..each** loop cannot be used if one specific value needs to be accessed for processing.

Java actually has two types of arrays. In this chapter, we learned about *static arrays*. Once a static array is constructed, you cannot change its size. The size of a static array can be found with the **length** attribute. **length** is like **Math.PI** and cannot be altered. Chapter 13 will introduce the **ArrayList** class which is a *dynamic array*. This means, you will be able to change the size of an **ArrayList**.

Static array can be used for one or more dimensions. A separate set of index operator brackets must be used for each index. Efficient access of a two-dimensional array requires a set of nested **for** loops.

A two-dimensional array is really a *one-dimensional array of one-dimensional arrays*. This is seen with the use of the <for.each> loop with two-dimensional arrays and with the **length** field when it is used with a two-dimensional array. The *ragged* array was introduced to demonstrate proper use of the **length** field when using two-dimensional arrays.

Static arrays store information differently than primitive data types. Primitive data types, like **int** and **double** store information in immediate, *shallow* memory locations. A data structure like an array only stores a memory reference to another location in its *shallow* memory. The array elements are stored in *deep* memory locations.

This difference is significant with parameter passing. Primitive data types, when passed to a method find that there is no impact on the values of the *actual* parameters as a consequence of calling any method.

On the other hand when a static array parameter is passed, the array elements of the *actual* parameter static array can be altered as a consequence of calling some method. This happens, because both the *actual* parameter and the *formal* parameter of the static array both reference the same *deep* memory locations where the static array elements are stored.