Array LOADING METHODS

# INTRODUCTION

I trust you have puzzled through the previous Tutorial on the basics of arrays. Now, we want to get down to serious business, literally! A great many business programs, as previously discussed, depend on having data fed to them from files that store up-to-date information on interest rates, spending data, customer data, etc. The possibilities are endless. This data changes so often that the only way for the program to do its job is to read the data from a file every time it executes, which is normally once a day.

How do we store this data when the program reads it in? The most common choice is some form of data structure that keeps what we need handy all the time the program is running. In this course our solution will always involve an array (hope that was obvious by now!).

We’ll start out by discussing exactly what it means to load an array, and then continue on with developing a complete solution that will load an array of employee ID numbers for use by the program.

# TWO METHODS TO LOAD AN ARRAY

There are two very common methods or times to load arrays with data: the first is to include the array’s contents in the code itself, sometimes called a **compile-time array**; and the second is to calculate the array’s elements or load them from a file, called a **run-time** or **execution-time array**.

**Compile-time arrays**…are very useful when you have similar values that don’t change, and you need to have them handy. Examples include data from all walks of life. Perhaps your company has a set of titles that you need to use in your program. They could be personnel titles or maybe just names of different subdivisions of the company. Maybe you need to use the names of various departments, for example, Accounting, Finance, Personnel, Management, Shipping, Inventory, etc. If you needed to use the department names often, you could place them into a String array.

Here would be one way to create an array of these names, using a named constant type of array:

final String[] DEPTS = {"Accounting", "Finance", "Personnel", "Management", "Shipping", "Inventory"};

Here I have tried to illustrate several different concepts. First, the values for the elements must be placed inside curly braces and separated by commas. Second, since we are specifying every element value, there is no need to use the new operator and code in the size of the array. The compiler will count the values and make the array that size. Last, since this list is not subject to change within the program, we have declared the array as a named constant, using the key word *final* in front of the declaration (and we used upper case for the name).

So, any time we need to refer to the Accounting department, we can just code DEPTS[0]. How about the Management department? Don’t forget that the list starts with zero, so the Management department would be DEPTS[3].

An important consideration for compile-time arrays is that all element values must be specified in the code itself…usually called “hard-coded”. As a result, any changes to the array would involve modifying the code and recompiling. This would be an important concern for any major program: you don’t want to go about recompiling it all the time! So, again, we restrict this sort of array for data that seldom, if ever, changes.

Here are more examples of initializing arrays in the source code.

int[] ageLimits = {16, 18, 21, 30, 65, 78}; //An array of integers

final char[] grades = {'A', 'B', 'C', 'D', 'F'}; //An array of chars

**Execution-time arrays**…are the proper choice for any arrays that will change during program execution or for data that must, by necessity, be loaded from some external source. Probably, the most common use of execution-time arrays in business programming is for loading data from a source file. This is quite common because business programs regularly do things such as prepare reports, crunch data, compute payroll, update inventory, etc., and the processing relies on sets of values that change from time to time.

For example, let’s suppose you are working on a program to update savings accounts at a bank. Such a program might run once a month to compute and record the interest on people’s accounts. Where would the program get the proper interest rates? The answer would most likely be that the program would need to open a file of rates and read them, storing the rates for use while the program executed. And the best solution for storing them would be an array.

Why couldn’t you just code the interest rates directly into the program? Certainly, you don’t want to have to recompile the program every month, just so you could recode the applicable interest rates! That would be a very poor design decision: once a program works, you don’t want to have to change it and recompile it at all.

So, it is very common to read data from files and load them into your program. We’ll spend the rest of this discussion learning how to do just that: load an important list of numbers into an array.

# Declaring arrays in a Java class. After our conceptual discussion of arrays, now we need to get back to the central topic of the class, OOP. We will be declaring arrays as part of our instance variables in a class that needs to store that data. And then we’ll need to develop methods to deal with those arrays. So, for example, we might declare a few arrays in the class this way:

# private int[] empNums = new int[1000];

# private double[] sales = new double[500];

# private char[] options = new char[5];

Next, let’s discuss how to load an array like the empNums array declared above. We’ll start by discussing the complete logic, then go on to implementing our logic in a Java class.

# LOADING AN ARRAY OF ID NUMBERS

Suppose we are working on a program—maybe a company payroll program—and we need a master list of employee ID’s to make sure we process only valid records. As we process each payroll record, we know it is very important to verify the identity of the individual. How can we do that?

A solid solution would be to create a list of the employees’ ID numbers from a master file of employee data. Then, as each payroll record is processed, compare the payroll ID against the master list. This type of security check is an important part of all business processing.

This raises two issues: how do we load the list; and how do we check each ID? We’ll handle the loading here and cover techniques for checking them in a later tutorial.

You should have found the project called “Acme Payroll Project” zipped up with this tutorial. Be sure to unzip it and then import it into Eclipse in order to get the most out of this discussion.

Assume, then, that we have access to a master employee personnel file (employees.dat), and also that we have the authority to read this file. We are going to pretend that we are interested in developing an entire payroll program. However, in this tutorial we’ll develop the class only as far as loading a master list of employees into an array. In order for this array to be available to all methods in the class, we’ll declare an array instance variable that is suitable to hold the ID numbers. We will also need a count variable that will hold the actual number of IDs loaded into the array. The array will be called empNums, and the counter will be called empCount.

Let’s make sure you understand how both variables are so important to each other. The array of employee IDs, empNums, will need to be declared large enough to hold everyone’s number in the file. So, we’ll need to declare the array to be a bit “over-size”. That’s forced on us by using Java arrays that need to be declared to be of a certain size. In other words, we’ll declare the array this way:

private int[] empNums = new int[1000];

As you have studied, the size needs to be set in the declaration statement. A potential alternative would be to open and read each record, and then store the number of records in a variable which could then be used to declare the array for the exact size. That would require two passes through the file, not a good choice for normal file processing.

Now, since we are going to have unused elements in the array, how can we know exactly how many elements are valid numbers loaded from the file? Answer: that’s what our empCount variable is for. We’ll count the number of elements we load using that variable, and as a result, it will be a very important part of the class for any further processing.

The actual loading of the array empNums will be accomplished in a class method called loadEmpNums. That method will be the only place that we deal with the file itself. That’s an important design decision for several reasons. Probably the most important is that if anything ever happens to the file, the only place we need to look for trouble is inside this method.

Following is the entire class source code.

public class Payroll

{

private int[] empNums = new int[1000];

private int empCount = 0;

Payroll(){} //Currently nothing done in constructor

public void loadEmpNums(String filename) // filename passed from main

{

String lastName, firstName;

char midInit;

double salary;

empCount = 0; //Just to make sure!

try

{

Scanner infile = new Scanner (new FileInputStream(filename));

while (infile.hasNext())

{

//Read a complete record

empNums[empCount] = infile.nextInt();

lastName = infile.next();

firstName = infile.next();

midInit = infile.next().charAt(0);

salary = infile.nextDouble();

//Increment the count of elements

++empCount;

}

infile.close();

}

catch (IOException ex)

{

//If file has problems, set the count to -1

empCount = -1;

ex.printStackTrace();

}

}//END loadEmpNums

public int getEmpCount()

{

return empCount;

}

public int getOneEmpNum(int index)

{

if (index >= 0 && index < empCount)

return empNums[index];

else

return -1;

}

}//END class Payroll

I have declared the array of employee IDs as an int merely to simplify your thought process in studying the example here. More likely, the array would need to be a String array, since ID’s commonly have non-digit characters in them.

Perhaps a major design decision is evident after looking at the declaration of the instance variables. It may not occur to you that a blank constructor is a design decision, but consider this: if and when a programmer creates a Payroll class object, when should the array of employee numbers be loaded? Some would argue that the array should be loaded right away, with no prompting from the application (main). There would be a couple of ways to accomplish that. One technique would be to place the entire loading logic into the constructor, meaning that as soon as it ran, the array would be good to go. Another technique would be to use the method we are using, the method loadEmpNums, but call it from the constructor. One method can call another with ease. That would also ensure that the array was ready as soon as the constructor finished executing.

Instead, this program expects the application, that is, the main method, to call loadEmpNums. I have done that for a couple of reasons, none of which are important for our discussion here. But you can see that, right after declaring variables in the main method, we do call the loader from there. Note that the filename is passed to the loadEmpNums method. By passing the file name (either here, or in some cases to the constructor) rather than putting the filename into the class itself we make the class more flexible.

Inside the loadEmpNums method, then, we’ll start by initializing the counter empCount to zero. This may not be all that essential, since we can also initialize it where it is declared. But it cannot hurt to make darn sure that 0 is its starting value. As you will see, it has a dual purpose in the loading process.

Next, we open the file by instantiating a Scanner class object. Like the file project we worked on earlier, we need to use a try/catch block to be able to catch any problems that may occur. In fact, if you attempt to skip using a try/catch block, Eclipse will complain about it and require you to correct the situation. If a problem does occur, we need some way to inform the application that it cannot proceed. The technique used here is to set the variable empCount to -1 in the catch block. That way, the main method could call getEmpCount and check the value for that error indication.

As long as the file did open, then we start the while loop using our look-ahead method hasNext. Inside the loop the fields we are reading from each record are all listed here, and you might argue that this list is a bit short (and I would agree!). There are probably tens of fields in a master employee data file. At any rate, these will be the five variables we’ll use to input a complete record from our data file. Obviously, I have kept the record size small for this example.

Also, notice that when it comes to reading in the employee number field, where are we placing the data? Directly into the array itself! Does it get to the right spot? The destination for the first employee number looks to be the first element of the array, empNums[0], since right now the counter is 0. So, that makes sense. Then, we read the other four fields from this record. Why read them if we really aren’t interested in them? The answer is that since this is a sequential access file, we have to keep reading data to get through it all. So if we read only the employee’s ID number, the next time we went to the file, we’d still be “staring at” that employee’s last name. The basic idea is this: any time you input from a sequential access file, always read a complete record.

At the bottom of the loop we add one to the variable empCount and then simply head on back to the top of the loop to see if there is more data in the file to read. There is no other processing to do right now—the only goal here is to load all the numbers into the array. In fact, it is really not a good idea to do any processing here: loading the array is a very important step, and should be divorced from any other processing. We’ll create other methods to process the data in the array.

An important note about the last statement in the loop: ++empCount; That statement actually accomplishes two important goals in the loading process. First, by incrementing its value, we have counted the record that we just read. Quite obvious. But at the same time we have “moved it along” so that at the top of the loop, the reference to empNums[empCount] will now address the next element in the array. Two birds with one stone!

Once we do reach EOF, we leave the while loop; close out the file; and return to the caller. Note carefully that we close the file as soon as we can: this is a master file that may be needed by other running programs. We have all we want from the file—a list of the ID numbers it contains—so, closing it right away makes it available for anyone else that wants it.

**Desk check the logic.** Now, are we sure that this logic will load the array correctly and return the exact count of numbers loaded? A short desk check will prove this. Assume that the company is very small, possibly a startup company, and currently there are only three employees to pay. Why write a computer program then? Obviously, we are planning to expand to a worldwide, billion-dollar company, so why not get the program perfected now, when it is easier to develop?!

Suppose the master file contains only those three employees and our program is ready to execute. The array is currently empty. The method loadEmpNums starts up, with the empCount being 0, of course.

empNums[ ]

|  |
| --- |
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |

empCount: 0

Inside loadEmpNums we open the file and read the first record. After reading the first record, the first array element contains the first ID, and we add one to empCount. At the bottom of the loop, here is the current state of our array and counter.

empNums[ ]

|  |
| --- |
| 987654321 |
| 0 |
| 0 |
| 0 |
| 0 |

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |

empCount: 1

Continuing the desk check, we read the second record and add 1 to empCount again. Now, our variables are updated with the second record’s information.

empNums[ ]

|  |
| --- |
| 987654321 |
| 123456789 |
| 0 |
| 0 |
| 0 |

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |

empCount: 2

Time to read the third and last record. The last record comes in, with the third ID placed into element 2. We add 1 to empCount once again.

empNums[ ]

|  |
| --- |
| 987654321 |
| 123456789 |
| 0 |
| 0 |
| 0 |

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |

empCount: 3

The hasNext method now tells us there is nothing further in the file to input, so the while loop shuts down. The array has all three IDs stored in it, and the counter empCount accurately tells us that we have a total of 3 good elements in the array.

From now on, any method using the array needs to keep that in mind that the counter will tell it how far to go in accessing the various elements. And we need to remember that the actual elements occupy positions from Ø up to the count minus 1. In the simple case above, you can easily tell that the array elements were loaded from subscripts Ø – 2, or 3 elements. So, any processing of the array must now take this into account and never attempt to process the countth element!

In the main method, then, we instantiate one object called acmePay, and then call the loading method. Following that, as a sort of test, I allow the user to input one index number and display the corresponding value from the array.

You should be able to use this project as the starting point for your next program. Be sure to go through the entire process of creating a new project, package, classes, etc. It’s very good practice…you don’t want to get rusty and forget any of the important steps.

**Note on the text**. The author discusses a nice way to process an array that does not require you to set up a counter to loop through all the array’s indices. This is called an **enhanced for loop**, and the syntax allows you to grab each element one at a time for processing. For example:

for (int value : empNums)

System.out.printf ("%d\n", value);

The variable *value* takes on every element of the array, one at a time, starting with element 0. This technique makes it a snap to traverse the array from the beginning all the way to the end. Unfortunately, this assumes that the entire array contains useful data, and in our case, it does not. The only useful numbers are in elements 0 – empCount-1. So, this technique has limited value in your programs for this course.

This concludes our discussion of loading an array from a data file. You will find many times throughout the course that you will need to load some data for later use. An array is an extremely useful tool when there is a lot of data to store, and especially, when you cannot know from day to day how much data there will be each time the program runs.

### End of Tutorial