CPS109 Course Notes 7

Alexander Ferworn

Unrelated Facts Worth Remembering

- The most successful people in any business are usually the most interesting.
- Don't confuse extensive documentation of a situation with insight, and don't confuse spreadsheets with analysis.
- Being certified in a skill associated with a product sold by a corporation is like being an unpaid salesperson for that corporation. Think about that when you are tempted by Microsoft certification.

Table of Contents

1 Introduction

Arrays are typically the first data structure or abstract data type (ADT) that most people run into in computer science. This document deals with them.

2 Data Structures

Definition:

- A data structure is an organized collection of related data.

An object is a type of data structure (although it is in fact more than this, since it also includes operations or methods for working with that data). However, this type of data structure -- consisting of a fairly small number of named instance variables -- is only one of the many different types of data structure that a programmer might need. In many cases, the programmer has to build more complicated data structures by linking objects together. But there is one type of data structure that is so important and so basic that it is built into every programming language: **the array**.

3 Arrays

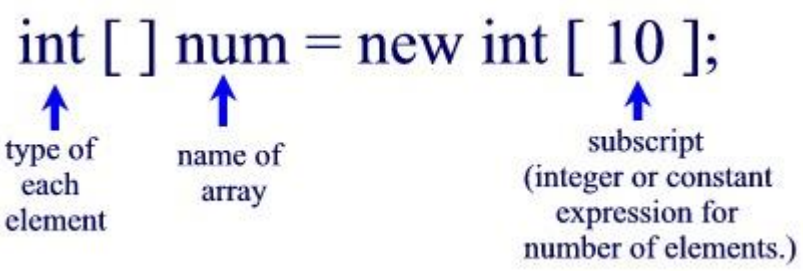
Definition

- An array is a data structure consisting of a numbered list of items, where all the items are of the same type.

In Java, the items in an array are always numbered from zero up to some maximum value. For example, an array might contain 100 integers, numbered from zero to 99. The items in an array can be objects, so that you could, for example, make an array containing all the Buttons in an applet. An array is our first example of a data structure.

3.1 Declaring an Array

Arrays are declared as any other variable in Java except that the declaration must also specify how many items can be stored in the array.



3.2 Records

Data structures can be quite complicated, but in many cases, a data structure consists simply of a sequence of data items. Data structures of this simple variety can be either arrays or records.

The term "record" is not used in Java. A **record** is essentially the same as a Java object that has instance variables only, but no instance methods. Some other languages, which do not support objects in general, do support records. The data items in a record -- in Java, its instance variables -- are called the **fields** of the record. Each item is referred to using a field name. In Java, field names are just the names of the instance variables. The distinguishing characteristics of a record are that the data items in the record are referred to by name and that different fields in a record are allowed to be of different types. For example, if the class Person is defined as:

```
class Person
{
     String name;
     int id_number;
     Date birthday;
     int age
```

```
}
```

then an object of class Person could be considered to be a record with four fields. The field names are

```
name, id_number, birthday, and age.
```

Note that the fields are of various types: String, int, and Date.

Like a record, an array is just a sequence of items. However, where items in a record are referred to by name, the items in an array are numbered, and individual items are referred to by their position number. Furthermore, all the items in an array must be of the same type. So we could define an array as a numbered sequence of items, which are all of the same type.

Definitions

- The number of items in an array is called the length of the array.
- The position number of an item in an array is called the index of that item.
- The type of the individual elements in an array is called the base type of the array.

In Java, items in an array are always numbered starting from zero. That is, the index of the first item in the array is zero. If the length of the array is N, then the index of the last item in the array is N-1. Once an array has been created, its length cannot be changed.

3.3 Arrays as Objects

Java arrays are objects. This has several consequences.

- Arrays are created using the new operator.
- No variable can ever hold an array; a variable can only refer to an array.
- Any variable that can refer to an array can also hold the value null, meaning that it doesn't at the moment refer to anything.
- Like any object, an array belongs to a class, which like all classes is a subclass of the class Object.

3.4 Using Arrays

Nevertheless, even though arrays are objects, there are differences between arrays and other types of objects, and there are a number of special language features in Java for creating and using arrays.

Suppose that A is a variable that refers to an array. Then the item at index k in A is referred to as

```
A[k]
```

The first item is A[0], the second is A[1], and so forth. "A[k]" can be used just like a variable. You can assign values to it, you can use it in expressions, and you can pass it as

a parameter to subroutines. All of this will be discussed in more detail below. For now, just keep in mind the syntax for referring to an item in an array.

Syntax:

- An array element is referred to as,

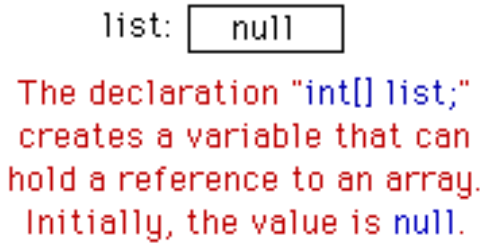
```
<array-variable> [ <integer-expression> ]
```

Although every array is a member of some class, array classes never have to be defined. Once a type exists, the corresponding array class exists automatically. If the name of the type is BaseType, then the name of the associated array class is BaseType[]. That is, an object belonging to the class BaseType[] is an array of items, where each item is a value of type BaseType. The brackets, "[]", are meant to recall the syntax for referring to the individual items in the array. "BaseType[]" can be read as "array of BaseType."

The base type of an array can be any legal Java type, that is, it can be a primitive type, an interface, or a class. From the primitive type int, the array type int[] is derived. Each item in an array of type int[] is a value of type int. From a class named Shape, the array type Shape[] is derived. Each item in an array of type Shape[] is a value of type class, which can be either null or a reference to an object belonging to the class Shape. (It might be worth mentioning here that if ClassA is a subclass of ClassB, then ClassA[] is automatically a subclass of ClassB[].)

Let's try to get a little more concrete about all this, using arrays of integers as an example. Since int[] is a class, it can be used to declare variables. For example,

```
...
int[] list;
...
```



This creates a variable named list of type int[].

This variable is capable of referring to an array of ints, but initially its value is null. The new operator can be used to create a new array object, which can then be assigned to list.

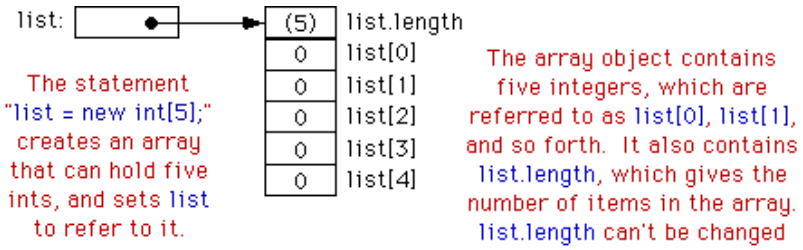
The syntax for using new with arrays is different from the syntax you learned previously for regular classes.

As an example,

```
...
list = new int[5];
...
```

creates an array of five integers. More generally, "new BaseType[N]" creates an array belonging to the class BaseType[]. The value N in brackets gives the length of the array, that is, the number of items that it holds.

Note that the array "knows" how long it is. The length of the array is something like an instance variable of the array object, and in fact, the length of the array list can be referred to as `list.length`. (However, you are not allowed to change the value of list.length, so its not really a regular instance variable.) The situation produced by the statement "list = new int[5];" can be pictured like this:



Note that the newly created array of integers is automatically filled with zeros. In Java, a newly created array is always filled with a known, default value.

The items in list are referred to a list[0], list[1], list[2], list[3], and list[4]. (Note that the index for the last item is one less than list.length.) However, array references can be much more general than this. The brackets in an array reference can contain any expression whose value is an integer. For example if indx is a variable of type int, then

```
list[indx] and
list[2*indx+7]
```

are syntactically correct references to items in the array list. And the following loop would print all the integers in the array, list, to standard output:

```
...
for (int i = 0; i < list.length; i++)
{
     System.out.println( list[i] );
}
...
```

The first time through the loop, i is 0, and list[i] refers to list[0]. The second time through, i is 1, and list[i] refers to list[1]. The loop ends after printing list[4], when i becomes equal to 5 and the continuation condition "i<list.length", is no longer true. This

is a typical example of using a loop to process an array. I'll discuss more examples of array processing in the next section.

If you think for a moment about what the computer will do when it encounters an expression such as "list[k]" while it is executing a program, you'll see that there are two things that can go wrong.

1. The expression is an attempt to access some specific element in the array referred to by the variable list. But suppose that the value of list is null. If that is the case, then list doesn't even refer to an array, and so the attempt to use the array item list[k] is an error. This is an example of a "**null pointer error**."
2. Even if list does refer to an array, it's possible that the value of k is outside the legal range of indices for that array. This will happen if k < 0 or if k >= list.length. In that case, once again, there is no such thing as list[k]. This is called an "**array index out of bounds**" error. When you use arrays in a program, you should be careful not to commit either of these errors.

For an array variable, just as for any variable, you can declare the variable and initialize it in a single step. For example,

```
int[] list = new int[5];
```

The new array is filled with the default value appropriate for the base type of the array -- zero for int and null for class types, for example. However, Java also provides a way to initialize an array variable with a new array filled with a specified list of values. This is done with an array initializer. For example,

```
int[] list = { 1, 4, 9, 16, 25, 36, 49};
```

creates a new array containing the seven values 1, 4, 9, 16, 25, 36, and 49, and sets list to refer to that new array. The value of list[0] will be 1, the value of list[1] will be 2, and so forth. The length of list is seven, since seven values are provided in the initializer.

An array initializer takes the form of an array of values, separated by commas and enclosed between braces. The length of the array does not have to be specified, because it is implicit in the list of values. The items in an array initializer don't have to be constants. They can be variables or expressions, provided that their values are of the appropriate type. For example, the following declaration creates an array of eight Colors. Some of the colors are given by expressions of the form "new Color(r,g,b)":

```
Color[] palette =
{
     Color.black,
     Color.red,
     Color.pink,
     new Color(0,180,0), // dark green
     Color.green,
```

```
            Color.blue,
            new Color(180,180,255), // light blue
            Color.white
      }
```

One final note: For historical reasons, the declaration

```
int[] list;
```

can also be written as

```
int list[];
```

which is a syntax used in the languages C and C++. However, this alternative syntax does not really make much sense in the context of Java, and it is probably best avoided. After all, the intent is to declare a variable of a certain type, and the name of that type is "int[]". It makes sense to follow the "type-name variable-name;" syntax for such declarations.

4 Array Processing

Arrays are the most basic and most important type of data structure, and techniques for processing arrays are among the most important programming techniques you can learn. Two fundamental array processing techniques -- searching and sorting -- will be covered in the next section.

In many cases, processing an array means applying the same operation to each item in the array. This is commonly done with a for loop. A loop for processing all the items in an array A has the form:

```
// do any necessary initialization
for (int i = 0; i < A.length; i++)
{
      . . .
      // process A[i]
      . . .
}
```

Suppose, for example, that A is an array of type double[]. Suppose that the goal is to add up all the numbers in the array. An informal algorithm for doing this would be:

Start with 0;
Add A[0]; (process the first item in A)
Add A[1]; (process the second item in A)
. . .
Add A[A.length - 1]; (process the last item in A)

Putting the obvious repetition into a loop and giving a name to the sum, this becomes:

```
double sum = 0;  // start with 0
for (int i = 0; i < A.length; i++)
     sum = sum + A[i];   // add A[i] to the sum, for
                 // i = 0, 1, ..., A.length - 1
```

Note that the continuation condition, "i < A.length", implies that the last value of i that is actually processed is A.length-1, which is the index of the final item in the array. It's important to use "<" here, not "<=", since "<=" would give an array out of bounds error. Eventually, you should almost be able to write loops similar to this one in your sleep. I will give a few more simple examples.

Here is a loop that will count the number of items in the array A which are less than zero:

```
int count = 0;  // start with 0 items counted
for (int i = 0; i < A.length; i++)
{
      if (A[i] < 0.0)   // if this item is less than zero..
      count++;          //     ...then count it
}     // At this point, the value of count is the number
      // of items that have passed the test of being < 0
```

Replace the test "A[i] < 0.0", if you want to count the number of items in an array that satisfy some other property. Here is a variation on the same theme. Suppose you want to count the number of times that an item in the array A is equal to the item that follows it. The item that follows A[i] in the array is A[i+1], so the test in this case is "A[i] == A[i+1]". But there is a catch: This test cannot be applied when A[i] is the last item in the array, since then there is no such item as A[i+1]. The result of trying to apply the test in this case would be an array out of bounds error. This just means that we have to stop one item short of the final item:

```
int count = 0;
for (int i = 0; i < A.length-1; i++)
{
      if (A[i] == A[i+1])
      count++;
}
```

Another typical problem is to find the largest number in A. The strategy is to go through the array, keeping track of the largest number found so far. We'll store the largest number found so far in a variable called max. As we look through the array, whenever we find a number larger than the current value of max, we change the value of max to that larger value. After the whole array has been processed, max is the largest item in the array overall. The only question is, what should the original value of max be? It makes sense to start with max equal to A[0], and then to look through the rest of the array, starting from A[1], for larger items:

```
double max = A[0];
for (int i = 1; i < A.length; i++)
{
     if (A[i] > max)
     max = A[i];
}    // at this point, max is the largest item in A
```

(There is one subtle problem here. It's possible in Java for an array to have length zero. In that case, A[0] doesn't exist, and the reference to A[0] in the first line gives an array out of bounds error. However, zero-length arrays are something that you want to avoid in real problems. Anyway, what would it mean to ask for the largest item in an array that contains no items at all?)

As a final example of basic array operations, consider the problem of copying an array. To make a copy of our sample array A, it is not sufficient to say

```
double[] B = A;
```

since this does not create a new array object. All it does is declare a new array variable and make it refer to the same object to which A refers. (So that, for example, a change to A[i] will automatically change B[i] as well.) To make a new array that is a copy of A, it is necessary to make a new array object and to copy each of the individual items from A into the new array:

```
double[] B = new double[A.length];
// make a new array object, the same size as A
for (int i = 0; i < A.length; i++)
     B[i] = A[i];   // copy each item from A to B
```

4.1 Partially Full Arrays

Once an array object has been created, it has a fixed size. Often, though, the number of items that we want to store in an array changes as the program runs. Since the size of the array can't actually be changed, a separate counter variable must be used to keep track of how many spaces in the array are in use. (Of course, every space in the array has to contain something; the question is, how many spaces contain useful or valid items?)

Suppose, for example, that your program is a game, and that players can join the game and leave the game as it progresses. As a good object-oriented programmer, you would probably have a class named Player to represent each individual player in the game. A list of all players who are currently in the game could be stored in an array, playerList, of type Player[]. Since the number of players can change, you will also need a variable, playerCt, to record the number of players currently in the game. Assuming that there will never be more than 10 players in the game, you could declare the variables as:

```
Player[] playerList = new Player[10];
```

```
// max of 10 players
int      playerCt = 0;
// at the start, there are no players
```

After some players have joined the game, playerCt will be greater than 0, and the player objects representing the players will be stored in the array items playerList[0], playerList[1], ..., playerList[playerCt-1]. Note that the array item playerList[playerCt] is not in use. The procedure for adding a new player, newPlayer, to the game is simple:

```
playerList[playerCt] = newPlayer;
// put new player in next
// available spot in array
playerCt++;
// and increment playerCt to count the new player
```

There will be a problem, of course, if you try to add more than 10 players to the game. It's possible to allow an unlimited number of players if you are willing to create a new, larger array whenever you run out of space:

```
// add a new player, even if the current array is full
if (playerCt == playerList.length) // test if array is full
{
      Player[] temp = new Player[playerCt + 10];
      // make a bigger array
      for (int i = 0; i < playerCt; i++)
      temp[i] = playerList[i];
      // copy REFERENCES to player objects
      playerList = temp;
      // set playerList to refer to new array
      // (the old array will be garbage-collected)
}
// At this point, we know there is room in the array
playerList[playerCt] = newPlayer; // add the new player...
playerCt++;  //    ...and count it
```

Deleting a player from the game is a little harder, since you don't want to leave a "hole" in the array. Suppose you want to delete the player at index k in playerList. If you are not worried about keeping the players in any particular order, then one way to do this is to move the player from the last position in the array into position k and then to decrement the value of playerCt:

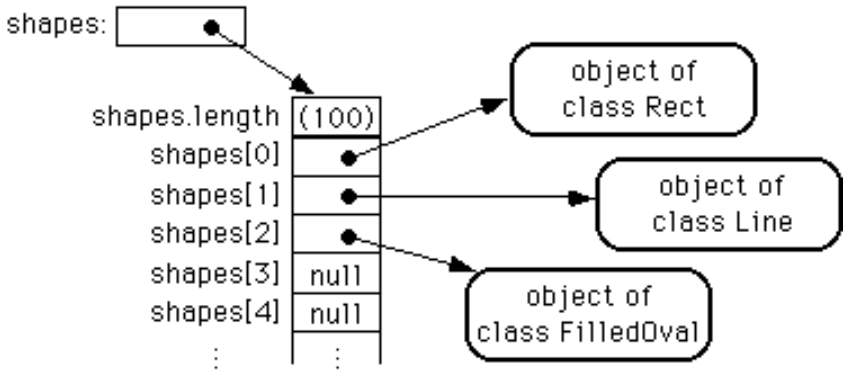
```
playerList[k] = playerList[playerCt - 1];          playerCt--;
```

It's worth emphasizing that the player example given above deals with an array whose base type is a class. An item in the array is either null or is a reference to an object belonging to the class Player. The Player objects themselves are not stored in the array. Note that because of the rules for assignment in Java, the objects can actually belong to subclasses of Player.

As another example, suppose that a class Shape represents the general idea of a shape drawn on a screen, and that it has subclasses to represent specific types of shapes such as lines, rectangles, rounded rectangles, ovals, filled-in ovals, and so forth. Then an array of type Shape[] can be declared and can hold references to objects belonging to the subclasses of Shape. For example, the situation created by the statements

```
Shape[] shapes = new Shape[100];
// array to hold 100 shapes
shapes[0] = new Rect();
// put some objects in the array
shapes[1] = new Line
//      (A real program would
shapes[2] = new FilledOval();
//       use some parameters here.)
int shapeCt = 3;
// keep track of number of objects in array
```

could be illustrated as:



Such an array could be useful in a drawing program. The array could be used to hold a list of shapes to be displayed. If the Shape class includes a redraw() method for drawing the shape, then all the shapes in the array could be redrawn with a simple for loop:

```
for (int i = 0; i < shapeCt; i++)
shapes[i].redraw();
```

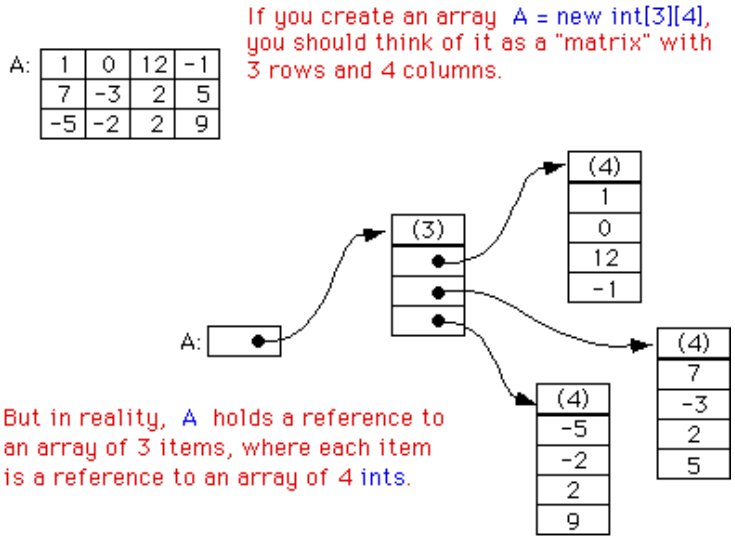
The statement "shapes[i].redraw();" calls the redraw() method belonging to the particular shape at index i in the array. Each object knows how to redraw itself, so that repeated executions of the statement can produce a variety of different shapes on the screen. This is nice example both of polymorphism and of array processing.

5 Two-Dimensional Arrays

Any type can be used as the base type for an array. You can have an array of ints, an array of Strings, or an array of Objects... And, since an array type is a first-class Java type, you can, in particular, have an array of arrays. For example, an array of ints is said

to have the type int[]. This means that there is automatically another type, int[][], which represents an "array of arrays of ints". Such an array is said to be a two-dimensional array. (Of course once you have the type int[][], there is nothing to stop you from forming the type int[][][], which represents a three-dimensional array -- and so on. However, in these notes I won't venture beyond the second dimension.)

The command "int[][] A = new int[3][4]" declares a variable, A, of type int[][], and it initializes that variable to refer to a newly created object. That object is an array of arrays of ints. The notation int[3][4] indicates that there are 3 arrays-of-ints in the array A, and that there are 4 ints in each of those arrays. However, trying to think in such terms can get a bit confusing -- as you might have already noticed. So it is customary to think of a two-dimensional array of items as a rectangular grid or matrix of items. The notation int[3][4] can then be taken to describe a grid of ints with 3 rows and 4 columns. The following picture might help:



For the most part, you can ignore the reality and keep the picture of a grid in mind. Sometimes, though, you will need to remember that each row in the grid is really an array in itself. These rows can be referred to as A[0], A[1], and A[2]. Each row is in fact an array of type int[]. It could, for example, be passed to a subroutine that asks for a parameter of type int[].

You can pick out a particular item from a row by adding another index. For example, A[1][3] refers to item number 3 in row number 1. Keep in mind, of course, that both rows and columns are numbered starting from zero. So, in the above example, A[1][3] is 5. More generally, A[i][j] refers to the int in row number i and column number j. The 12 items in A would be named as follows:

```
A[0][0]
A[0][1]
A[0][2]
```

```
A[0][3]
A[1][0]
A[1][1]
A[1][2]
A[1][3]
A[2][0]
A[2][1]
A[2][2]
A[2][3]
```

It might be worth noting that A.length gives the number of rows of A. To get the number of columns in A, you have to ask how many ints there are in a row; this number would be given by A[0].length, or equivalently by A[1].length or A[2].length. (There is actually no rule that says that the rows of an array must have the same length, and some advanced applications of arrays use varying-sized rows. But if you use the new operator to create an array in the manner described above, you'll get an array with equal-sized rows.)

It's possible to fill a two-dimensional array with specified items at the time it is created. Recall that when an ordinary one-dimensional array variable is declared, it can be assigned an "array initializer," which is just a list of values enclosed between braces, { and }. Similarly, a two-dimensional array can be created as a list of array initializers, one for each row in the array. For example, the array A shown in the picture above could be created with:

```
int[][]  A = {{1,0,12,-1},{7,-3,2,5},{-5,-2,2,9}};
```

If no initializer is provided for an array, then when it is created it is automatically filled with the appropriate value: zero for numbers, false for boolean, and null for objects.

A two-dimensional array can be used whenever the data being represented can be naturally arranged into rows and columns. Often, the grid is built into the problem. For example, a chess board is a grid with 8 rows and 8 columns. If a class named ChessPiece is available to represent individual chess pieces, then the contents of a chess board could be represented by a two-dimensional array

```
ChessPiece[][]  board  =  new ChessPiece[8][8];
```

Or consider the "mosaic" of colored squares used as an example previously. The data about the color of each of the squares in the mosaic is stored in an array of type Color[][]. If the mosaic has ROWS rows and COLUMNS columns, then the array of color data can be created with the statement

```
Color[][]  colorGrid  =  new Color[ROWS][COLUMNS];
```

When the color of the square in row i and column j is set to some color value c, the square is redrawn on the screen in the new color, and the color is stored in the array with an assignment "colorGrid[i][j] = c". The information in the colorGrid array can be used to redraw the whole mosaic when necessary. The information in the array is also used when

you want to find out the color at a particular location in the array. The assignment statement "c = colorGrid[i][j]" gets the color of the square in row i and column j.

The grid is not always so visually obvious in a problem. Consider a company that owns 25 stores. Suppose that the company has data about the profit earned at each store for each month in the year 1995. If the stores are numbered from 0 to 24, and if the twelve months from January '95 through December '95 are numbered from 0 to 11, then the profit data could be stored in an array, profit, constructed as follows:

```
double[][]  profit  =  new double[25][12];
```

profit[3][2] would be the amount of profit earned at store number 3 in March, and more generally, profit[storeNum][monthNum] would be the amount of profit earned in store number storeNum in month number month. In this example, the one-dimensional array profit[storeNum] has a very useful meaning: It is just the profit data for one particular store for the whole year.

Just as in the case of one-dimensional arrays, two-dimensional arrays are often processed using for statements. To process all the items in a two-dimensional array, you have to use one for statement nested inside another. If the array A is declared as

```
int[][]  A  =  new int[3][4];
```

then you could store a zero into each location in A with:

```
for (int row = 0;  row < 3;  row++)
{
      for (int column = 0;  row < 4;  column++)
      {
            A[row][column] = 0;
      }
}
```

The first time the outer for loop executes (with row = 0), the inner four loop fills in the four values in the first row of A, namely A[0][0] = 0, A[0][1] = 0, A[0][2] = 0, and A[0][3] = 0. The next execution of the outer for loop fills in the second row of A. And the third and final execution of the outer loop fills in the final row of A.

Similarly, you could add up all the items in A with:

```
int sum = 0;
for (int i = 0; i < 3; I
      for(int j = 0; j < 4; i++)
            sum = sum + A[i][j];
```

If we did the same thing with the profit array discussed above, this example might seem a little more interesting. The sum would be the total profit earned by the company over the course of the entire year in all of its twenty-five stores.

This profit example demonstrates that sometimes it is necessary to process a single row or a single column of an array. For example, to compute the total profit earned by the company in December, that is, in month number 11, you could use the loop:

```
double decemberProfit = 0.0;
for (storeNum = 0; storeNum < 25; storeNum++)
      decemberProfit += profit[storeNum][11];
```

We could extend this idea to create a one-dimensional array that contains the total profit for each month of the year:

```
double[] monthlyProfit = new double[12];
for (int month = 0; month < 12; month++)
{
      // compute the total profit from all stores
      // in this month
      monthlyProfit[month] = 0.0;
      for (int store = 0; store < 25; store++)
            monthlyProfit[month] += profit[store][month];
}
```

As a final example of processing two-dimensional arrays, suppose that we wanted to know which store generated the most profit over the course of the year. To do this, we have to add up the monthly profits for each store. In array terms, this means that we want to find the sum of each row in the array. As we do this, we want to keep track of which row produces the largest total.

```
double maxProfit;
// maximum profit earned by a store
int bestStore;
// the number of the store with the
// maximum profit
double total = 0.0;
// total profit for one store;
// first compute the profit from store number 0

for (int month = 0;  month < 12; month++)
      total += profit[0][month];
bestStore = 0;
// start by assuming that the best
maxProfit = total;
// store is store number 0
// Now, go through the other stores, and whenever you
// find one with a bigger profit than maxProfit, revise
// the assumptions about bestStore and maxProfit
for (store = 1; store < 25; store++)
{
      total = 0.0;
      for (month = 0; month < 12; month++)
            total += profit[store][month];
      if (total > maxProfit)
```

```
            {
                  maxProfit = total;
                  // best profit seen so far!
                  bestStore = store;
                  // and it came from this store
            }
      }
      // At this point, maxProfit is the best profit of any
      // of the 25 stores, and bestStore is a store that
      // generated that profit.  (Note that there could be
      // other stores that generated the same profit.)
```