Arrays

Motivation for Arrays

Suppose we want to store 100 values entered by the user. Knowing what we do so far, we would have to declare 100 different variables to hold the information. This would be doable, but it would be a giant pain.

Next, suppose we want to store a bunch of values entered by the user, but we don't know how many – it might be 10, it might be 100,000. We could try to declare 100,000 variables, just in case, but even this wouldn't work if the user decided to store more than that. To solve this problem, we need a convenient to store a list of values. Furthermore, we want this list to be able to hold any number of values, depending on the user's needs.

Java has a feature called an *array* that is used to store a variable number of elements.

Single-Dimensional Arrays

A *single-dimensional array* stores a list of elements of the same type. We can make this list be any size that we want.

Declaration

Here's how to declare an array:

```
type[] name;
```

This will declare an array called `name` that can hold elements of type `type`. For example, to declare an array of `int`s called `nums`:

```
int[] nums;
```

Now, we need to create space for the array. Before we can add any elements to the array, we need to specify the number of slots we want to reserve. Here's how:

```
name = new type[size];
```

Here, `name` is the name of the array, `type` is the type of elements the array holds, and `size` is a positive integer that specifies how many slots we want in the array. `size` can be either a constant value (like 10) or an `int` variable. Here's how to make our `nums` array have 10 slots:

```
nums = new int[10];
```

We can also declare an array and reserve space for it on the same line:

```
int[] nums = new int[10];
```

Initialization
Now that we have space for the array elements, we can start putting values in the array. Here's how to access array elements:

```
name[index]
```

This accesses the element in array `name` at index `index`. C# array indices start at 0 and go up to `size`-1 (where `size` is the number of slots we reserved for the array). For example, here's how we'd set the first element in the `nums` array to 7:

```
nums[0] = 7;
```

Here's how we'd set the last element in `nums` to 4:

```
nums[9] = 4;          //9 is the last index since the array size is 10
```

Examples
Arrays are very naturally processed with for loops. The loop counter is the array index – we start at 0, and we continue looping while the loop counter is less than the array size. Each time, we add one to the loop counter. For example, here's how we'd use a loop to set every element in the `nums` array to 0:

```
for (int i = 0; i < 10; i++) {
     nums[i] = 0;
}
```

We can also retrieve the size of the array with the command:

```
name.length
```

where `name` is the name of the array. For another example, suppose we want to ask the user for 10 numbers, and then we want to print them in reverse order. Here's how:

```
Scanner s = new Scanner(System.in);
int[] nums = new int[10];
for (int i = 0; i < nums.length; i++) {
     System.out.print("Enter a number: ");
     nums[i] = Integer.parseInt(s.nextLine());
}

for (int i = nums.length-1; i >= 0; i--) {
     System.out.println(nums[i]);
}
```

Nested Loops
When processing arrays, it is sometimes necessary to put a loop inside of another loop – this is called a *nested loop*.

Suppose we want to ask the user for 10 numbers, store them in an array, and then print them out in sorted order. There are many ways to sort numbers, but one of the easiest ways is to find the min, print it, find the next min, print it, etc., until all the numbers have been printed. We will have an outer loop that prints the next-smallest array element with each iteration. Each time the outer loop is processed, we will execute an inner loop that steps through all array elements to find the smallest one.

Here is the code:

```
//create an array to hold the numbers
int[] nums = new int[10];

//get the input numbers
Scanner s = new Scanner(System.in);
for (int i = 0; i < nums.length; i++) {
      System.out.print("Enter a positive number: ");
      nums[i] = Integer.parseInt(s.nextLine());
}

//the index of the minimum value in the array
int minIndex = 0;
int min = 0;

System.out.println("Sorted:");

//find the next-smallest number and print it
for (int i = 0; i < nums.length; i++) {

      //find the min
      for (int j = 0; j < nums.length; j++) {

            //is this the first available element?
            if (min == 0 && nums[j] != -1) {
                  minIndex = j;
                  min = nums[j];
            }
            //do we have a new min?
            else if (nums[j] < min && nums[j] != -1) {
                  minIndex = j;
                  min = nums[j];
            }
      }
      //print the min
      System.out.println(min);
```

```
        //set to -1 so it won't be found again
        nums[minIndex] = -1;
        min = 0;
        minIndex = 0;
    }
```

Again, every time the outer loop executes, we go through the entire array in the inner loop to look for the next-smallest number.

Two-Dimensional Arrays

The arrays we've seen so far have been one-dimensional – that is, just a list of elements. We can also create multi-dimensional arrays, which are really just arrays of arrays. For example, a 2-dimensional array could represent a mathematical matrix, where each element has an associated row and column.

Declaration

Here's how to declare a 2-dimensional array:

```
type[][] name;
```

Here's how to give the array space:

```
name = new type[numRows][numCols];
```

This creates a two-dimensional array called `name`, `numRows x numColumns`, that holds elements of type `type`. Here's how to create a 5x10 array of integers:

```
int[][] matrix = new int[5][10];
```

Initialization

To access elements in a two-dimensional array, you specify both the row number and the column number. Here's how to set the element at row 2 and column 0 to 6:

```
matrix[2][0] = 6;
```

Examples

If we want to initialize all elements of a two-dimensional array, we need to use a **nested for loop**. The outer loop will step through each row, and the inner loop will step through each column in that row. Here's how to use a two-dimensional array to represent a multiplication table (the entry at row `i` and column `j` has the `value i*j`):

```
//Declare, create space
int[][] multTable = new int[10][10];
```

```
//Fill with multiplication values
for (int i = 0; i < 10; i++) {
     for (int j = 0; j < 10; j++) {
          multTable[i][j] = i*j;
     }
}
```

Now the array looks like this:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

For-each Loop

There is a fourth kind of loop in Java that is handy for stepping through collections of elements, such as arrays. The syntax is:

```
for (type name : list) {

}
```

Here, `type` is the type of elements in the list, `name` is a variable that will step through each element, and `list` is the name of the list or array of elements.

To see how the for-each loop works, suppose we have declared this array:

```
double[] vals;
```

Further suppose that we have allocated space for the array and filled it with initial values. Here is how we could use a for-each loop to print every element in the array:

```
for (double num : vals) {
     System.out.println(num);
}
```

This loop will step through the array starting and index 0. Each time, `num` will be given the value of the next element in the array.

One caveat to using a for-each loop is that you CANNOT use it to change any values in the array. For example, we might try to set every value in the `vals` array back to zero:

```
for (double num : vals) {
     //Won't change array
     num = 0.0;
}
```

This code would compile, but it would not change the array. In that for-each loop, `num` is assigned the value of the first element in the array, then the second, etc. What the above loop does is change the current value of `num` – not the current array spot. If you want to change elements in an array, use one of the other types of loops.

We can also use a for-each loop to step through every element in a two-dimensional array. Just like with other loops, we need an outer loop to step through the rows, and an inner loop to step through the elements on that row. For example:

```
int count = 0;
int[][] arr = new int[4][5];
for (int i = 0; i < 4; i++) {
     for (int j = 0; j < 5; j++) {
          arr[i][j] = count;
          count++;
     }
}

/*Now the array looks like:
     0  1  2  3  4
     5  6  7  8  9
     10 11 12 13 14
     15 16 17 18 19
*/

//elements printed will be 0, 1, 2, etc. ("read" array like a book)
for (int[] val : arr) {
     for (int x : val) {
          System.out.println(x);
     }
}
```

More Examples

This section includes a few more examples of full programs that use arrays.

Example 1
Suppose we want to write a program that asks the user for 20 numbers (doubles), and that calculates and prints the average, minimum, and maximum number from the user.

```
import java.util.*;
public class Example1 {
     public static void main(String[] args) {
          Scanner s = new Scanner(System.in);
          double[] nums = new double[20];

          for (int i = 0; i < nums.length; i++) {
               System.out.print("Enter the next number: ");
               nums[i] = Double.parseDouble(s.nextLine());
          }

          //find the average
          double sum = 0;
          for (int i = 0; i < nums.length; i++) {
               sum += nums[i];
          }

          double average = sum / nums.length;
          System.out.println("Average: " + average);

          //find the minimum
          double min = nums[0];
          for (int i = 1; i < nums.length; i++) {
               if (nums[i] < min) min = nums[i];
          }

          System.out.println("Minimum: " + min);

          //find the maximum
          double max = nums[0];
          for (int i = 1; i < nums.length; i++) {
               if (nums[i] > max) max = nums[i];
          }

          System.out.println("Maximum: "  + max);
     }
}
```

It turns out that we don't need to write separate code to calculate the average, minimum, and maximum. We can combine those steps into a single loop. (We also don't *need* to store the numbers in an array before doing our calculations, but we'll keep that part since it's the focus of this chapter.) Here is a shorter way to write the same program:
import java.util.*;

```
public class Example1 {
     public static void main(String[] args) {
          Scanner s = new Scanner(System.in);
          double[] nums = new double[20];

          for (int i = 0; i < nums.length; i++) {
               System.out.print("Enter the next number: ");
               nums[i] = Double.parseDouble(s.nextLine());
          }

          //find the average, minimum, and maximum
          double sum = 0;
          double min = nums[0];
          double max = nums[0];

          //for the min/max, it doesn't *hurt* to start at i=0
          for (int i = 0; i < nums.length; i++) {
               sum += nums[i];
               if (nums[i] < min) min = nums[i];
               if (nums[i] > max) max = nums[i];
          }

          double average = sum / nums.length;
          System.out.println("Average: " + average);
          System.out.println("Minimum: " + min);
          System.out.println("Maximum: "  + max);
     }
}
```

Example 2

In this example, we will write a full program that reverses an array in place. We will start by feeling the array with default values (we could get user input for the array values – the next steps would be the same in either case). Then, we will write the code necessary to switch array elements around so that they're stored in reverse of the original order. Finally, we'll print the resulting array to show that it will print in reverse order.

```
public class Example2 {
     public static void main(String[] args) {
          //could be any values, or could have gotten user input
          int[] default = {2,4,6,8,10,12,14,16,20};

          //now reverse the order of the array elements
          int back = default.length-1;

          //front starts at the beginning of the array and steps back
          //back starts at the end of the array and steps up
```

```
            //why do we only go halfway?  Think about this.
            for (int front=0; front < default.length/2; front++) {
                  //at each step, swap the current front and back
                  int temp = default[front];
                  default[front] = default[back];
                  default[back] = temp;
            }

      }
}
```

Example 3

In this example, we will write a full Tic-Tac-Toe program between two players. We will use a 3x3 array of characters to store the board.

```
import java.util.*;
public class TicTacToe {
      public static void main(String[] args) {
            Scanner s = new Scanner(System.in);
            char[][] board = new char[3][3];

            //fill with _ for blank spots
            for (int i = 0; i < 3; i++) {
                  for (int j = 0; j < 3; j++) {
                        board[i][j] = '_';
                  }
            }

            //count how many moves have been made
            int moves = 0;

            //keep track of whose turn it is
            char turn = 'X';

            //print the initial board
            System.out.println("Current board: ");
            for (int i = 0; i < 3; i++) {
                  for (int j = 0; j < 3; j++) {
                        System.out.print(board[i][j] + " ");
                  }
                  System.out.println();
            }
            System.out.println();

            //keep playing while less than 9 moves
            while (moves < 9) {
```

```
System.out.print(turn + ", enter row: ");
int row = Integer.parseInt(s.nextLine());
System.out.print(turn + ", enter column: ");
int col = Integer.parseInt(s.nextLine());

//check to see if that is a valid move
if (row < 0 || row > 2 || col < 0 || col > 2) {
     System.out.println("invalid row/column");
}
else if (board[row][col] != '_') {
     System.out.println("That spot is taken");
}
else {
     board[row][col] = turn;
     //print the board
     System.out.println("\nCurrent board: ");
     for (int i = 0; i < 3; i++) {
          for (int j = 0; j < 3; j++) {
               System.out.print(board[i][j]+" ");
          }
          System.out.println();
     }
     System.out.println();

     //check for a winner
     boolean win = false;
     for (int i = 0; i < 3; i++) {
          if (board[i][0] == board[i][1] &&
               board[i][1] == board[i][2] &&
               board[i][0] == turn) win = true;

          if (board[0][i] == board[1][i] &&
               board[1][i] == board[2][i] &&
               board[0][i] == turn) win = true;
     }
     if (board[0][0] == board[1][1] &&
          board[1][1] == board[2][2] &&
          board[0][0] == turn) win = true;

     if (board[2][0] == board[1][1] &&
          board[1][1] == board[0][2] &&
          board[2][0] == turn) win = true;


     if (win) {
          System.out.println(turn + " wins!");
```

```
                        //end the game
                        break;
                    }

                    //switch whose turn it is
                    if (turn == 'X') turn = 'O';
                    else turn = 'X';

                    moves++;
                }
            }

            //if moves made it to 9, must be a tie
            if (moves == 9) System.out.println("Tie game.");
        }
}
```