Think Java

CHAPTER 7: ARRAYS AND REFERENCES

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Up to this point, the only variables we have used were for individual values such as numbers or strings. In this chapter, we'll learn how to store multiple values of the same type using a single variable. This language feature will enable you to write programs that manipulate larger amounts of data.
- For example, Exercise 5 asked you to check whether every letter in a string appears exactly twice. One algorithm (which hopefully you already discovered) is to loop through the string 26 times, once for each lowercase letter:

```
for (char c = 'a'; c <= 'z'; c++) {
    // count how many times the letter appears
    // if the count is not 0 or 2, return false
}
```

Objectives

- This “nested loops” approach is inefficient, especially when the string is long (e.g., one billion characters). Another algorithm would initialize 26 variables to zero, loop through the string once, and use a giant if statement update the variable for each letter. But who wants to declare 26 variables?
- That’s where arrays come in. We can declare a single variable that stores 26 integers. Rather than use an if statement to update each value, we can use arithmetic to update the nth value directly. We will present this algorithm at the end of the chapter.

Topics

- Declaration and Initialization of an Array.
- Copy and Traversal of an array
- Assigning Random Values to an Array
- Histogram
- Loop Operations over an Array
- Application of Histogram

LECTURE 1

Creating arrays

Declaration of An Array

- An array is a sequence of values; the values in the array are called elements. You can make an array of ints, doubles, Strings, or any other type, but all the values in an array must have the same type.
- To create an array, you have to declare a variable with an array type and then create the array itself. Array types look like other Java types, except they are followed by square brackets ([]). For example, the following lines declare that counts is an “integer array” and values is a “double array”:

```
int[] counts;
double[] values;
```

Instantiationof an Array

- To create the array itself, you have to use the new operator, which we first saw in Section 3.2. The new operator allocates memory for the array and automatically initializes all of its elements to zero.

```
counts = new int[4];
values = new double[size];
```

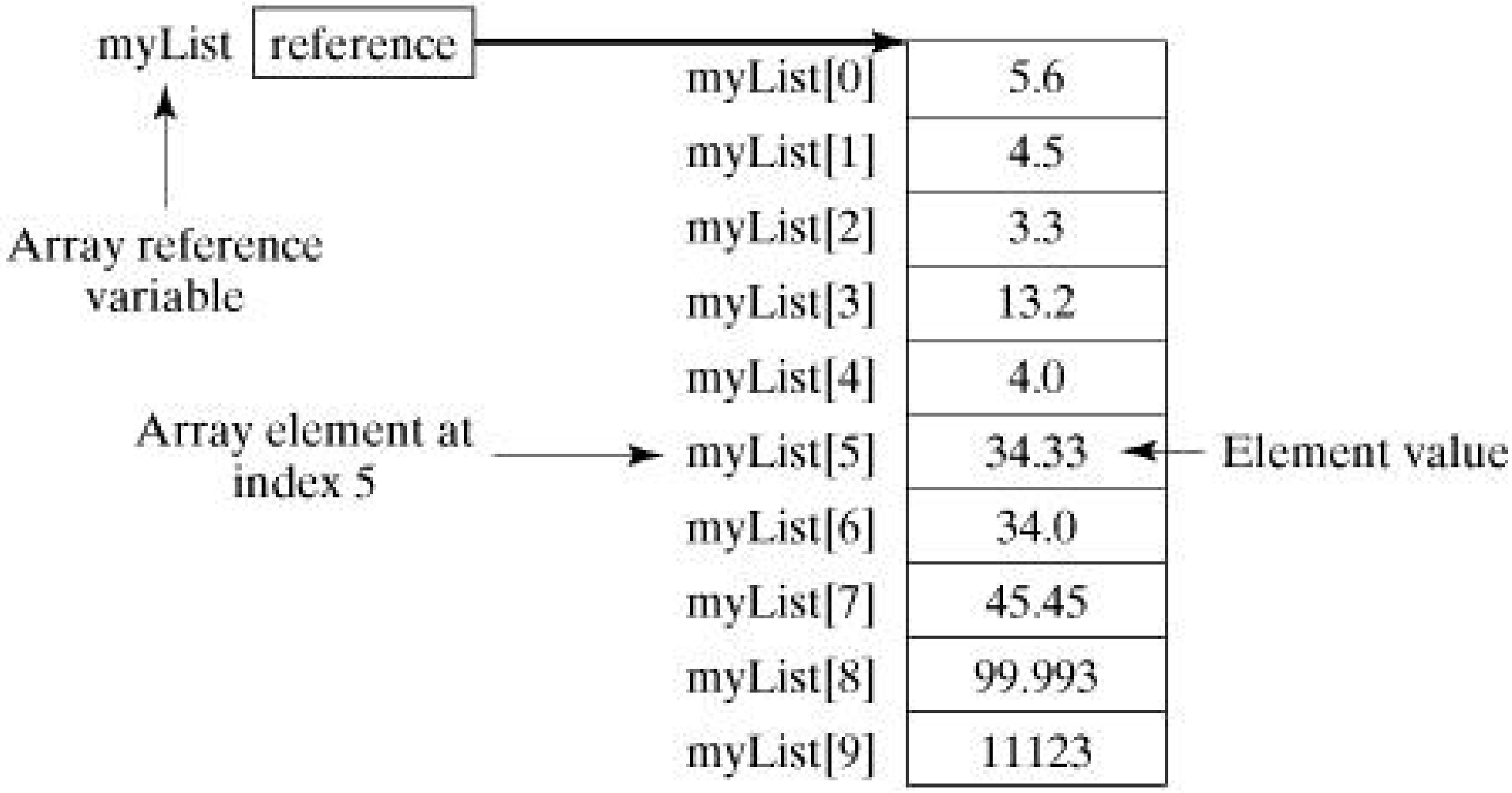
- The first assignment makes counts refer to an array of four integers. The second makes values refer to an array of doubles, but the number of elements depends on the value of size (at the time the array is created).

Declaration and Instantiation of an Array

- Of course, you can also declare the variable and create the array with a single line of code:

```
int[] counts = new int[4];
double[] values = new double[size];
```

- You can use any integer expression for the size of an array, as long as the value is nonnegative. If you try to create an array with -4 elements, for example, you will get a **NegativeArraySizeException**. An array with zero elements is allowed, and there are special uses for such arrays that we'll see later on.


myList
reference
Array reference variable
Array element at index 5
myList[0]
myList[1]
myList[2]
myList[3]
myList[4]
myList[5]
myList[6]
myList[7]
myList[8]
myList[9]
5.6
4.5
3.3
13.2
4.0
34.33
34.0
45.45
99.993
11123
Element value

LECTURE 2

Accessing elements

Initialization of Array Elements

- When you create an array with the new operator, the elements are initialized to zero. Figure 7.1 shows a memory diagram of the counts array so far.

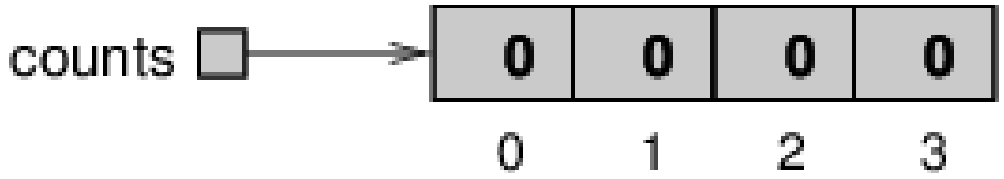


Figure 7.1: Memory diagram of an int array.

Array is of Reference Type

- The arrow indicates that the value of counts is a reference to the array. You should think of the array and the variable that refers to it as two different things. As we'll soon see, we can assign a different variable to refer to the same array, and we can change the value of counts to refer to a different array.
- The large numbers inside the boxes are the elements of the array. The small numbers outside the boxes are the indexes used to identify each location in the array. As with strings, the index of the first element is 0, not 1. For this reason, we sometimes refer to the first element as the "zeroth" element.

Indexing: Access of each Element

The [] operator selects elements from an array:

```
System.out.println("The zeroth element is " +
counts[0]);
```

You can use the [] operator anywhere in an expression:

```
counts[0] = 7;
counts[1] = counts[0] * 2;
counts[2]++;
counts[3] -= 60;
```

Indexing

- You can use any expression as an index, as long as it has type int. One of the most common ways to index an array is with a loop variable. For example:

```
int i = 0;
while (i < 4) {
  System.out.println(counts[i]);
  i++;
}
```

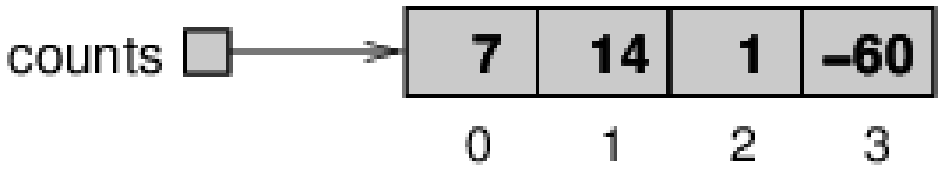


Figure 7.2: Memory diagram after several assignment statements.

ArrayIndexOutOfBoundsException

- This while loop counts up from 0 to 4. When i is 4, the condition fails and the loop terminates. So the body of the loop is only executed when i is 0, 1, 2, and 3. In this context, the variable name i is short for “index”.
- Each time through the loop we use i as an index into the array, displaying the ith element. This type of array processing is usually written as a for loop.

```
for (int i = 0; i < 4; i++) {
    System.out.println(counts[i]);
}
```

- For the counts array, the only legal indexes are 0, 1, 2, and 3. If the index is negative or greater than 3, the result is an **ArrayIndexOutOfBoundsException**.

LECTURE 3

Displaying arrays

hashCode of an Object

- You can use println to display an array, but it probably doesn't do what you would like. For example, the following fragment (1) declares an array variable, (2) makes it refer to an array of four elements, and (3) attempts to display the contents of the array using println:

```
int[] a = {1, 2, 3, 4};
System.out.println(a);
```

- Unfortunately, the output is something like:
  - `[I@bf3f7e0`

printArray (Array Traversal)

- The bracket indicates that the value is an array, I stands for “integer”, and the rest represents the address of the array in memory. If we want to display the elements of the array, we could do it ourselves:

```
public static void printArray(int[] a) {
    System.out.print("{" + a[0]);
    for (int i = 1; i < a.length; i++) {
        System.out.print(", " + a[i]);
    }
    System.out.println("}");
}
```

Print Array using Arrays.toString()

- Given the previous array, the output of printArray is:

```
{1, 2, 3, 4}
```

- The Java library provides a utility class java.util.Arrays that has methods for working with arrays. One of them, toString, returns a string representation of an array. We can invoke it like this:

```
System.out.println(Arrays.toString(a));
```

- And the output is:

```
[1, 2, 3, 4]
```

- As usual, we have to import java.util.Arrays before we can use it. Notice that the string format is slightly different: it uses square brackets instead of curly braces. But it beats having to write your own printArray method.

LECTURE 4

Copying arrays

Memory Diagram for Array

- As explained in Section 7.2, array variables contain references to arrays. When you make an assignment to an array variable, it simply copies the reference. But it doesn't copy the array itself. For example:

```
double[] a = new double[3];
double[] b = a;
```

- These statements create an array of three doubles and make two different variables refer to it, as shown in Figure 7.3.

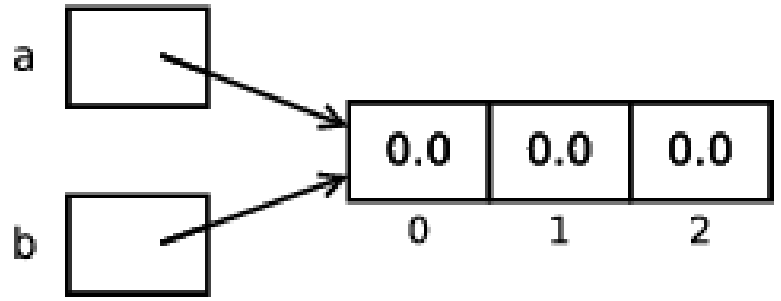


Figure 7.3: Memory diagram of two variables referring to the same array.

Reference: Aliases

- Any changes made through either variable will be seen by the other. For example, if we set a[0] = 17.0, and then display b[0], the result is 17.0. Because a and b are different names for the same thing, they are sometimes called **aliases**.
- If you actually want to copy the array, not just the reference, you have to create a new array and copy the elements from one to the other, like this:

```
double[] b = new double[3];
for (int i = 0; i < 3; i++) {
    b[i] = a[i];
}
```

Memory Diagram

- java.util.Arrays provides a method named copyOf that performs this task for you. So you can replace the previous code with one line:

```
double[] b =
Arrays.copyOf(a, 3);
```

- The second parameter is the number of elements you want to copy, so copyOf can also be used to copy part of an array. Figure 7.4 shows the state of the array variables after invoking Arrays.copyOf

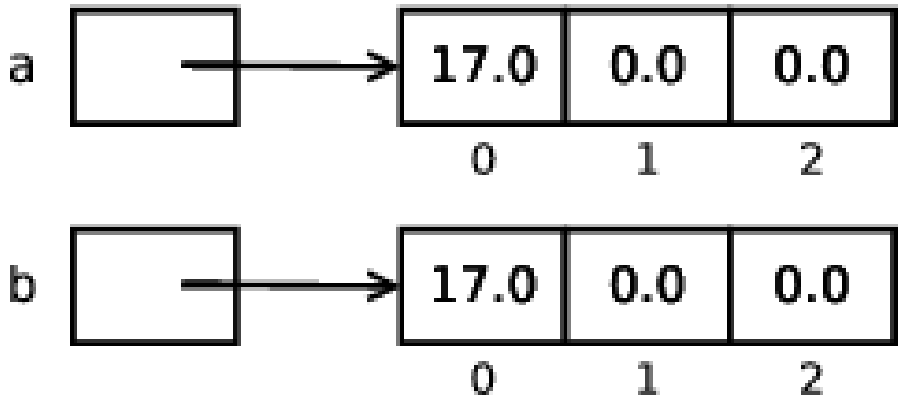


Figure 7.4: Memory diagram of two variables referring to different arrays.

Transcopy of Arrays

- The examples so far only work if the array has three elements. It is better to generalize the code to work with arrays of any size. We can do that by replacing the magic number, 3, with a.length:

```
double[] b = new double[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

- All arrays have a built-in constant, length, that stores the number of elements. In contrast to String.length(), which is a method, a.length is a constant. The expression a.length may look like a method invocation, but there are no parentheses and no arguments.

Using Arrays.copyOf()

- The last time the loop gets executed, i is a.length - 1, which is the index of the last element. When i is equal to a.length, the condition fails and the body is not executed – which is a good thing, because trying to access a[a.length] would throw an exception.
- Of course we can replace the loop altogether by using Arrays.copyOf and a.length for the second argument. The following line produces the same result shown in Figure 7.4.

```
double[] b = Arrays.copyOf(a, a.length);
```

LECTURE 5

Array Traversal

Array Traversal

- Many computations can be implemented by looping through the elements of an array and performing an operation on each element. Looping through the elements of an array is called a traversal.

```
int[] a = {1, 2, 3, 4, 5};
for (int i = 0; i < a.length; i++) {
    a[i] *= a[i];
}
```

- This example traverses an array and squares each element. At the end of the loop, the array has the values \{1, 4, 9, 16, 25\}.

Linear Search

- Another common pattern is a search, which involves traversing an array and "searching" for a particular element. For example, the following method takes an array and a value, and it returns the index where the value appears:

```
public static int search(double[] array, double
target) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == target) {
            return i;
        }
    }
    return -1;  // not found
}
```

Linear Search

- If we find the target value in the array, we return its index immediately. If the loop exits without finding the target, it returns -1, a special value chosen to indicate a failed search. (This code is essentially what the String.indexOf method does.)
- The following code searches an array for the value 1.23, which is the third element. Because array indexes start at zero, the output is 2.

```
double[] array = {3.14, -55.0, 1.23, -0.8};
int index = search(array, 1.23);
System.out.println(index);
```

Traversal

- Another common traversal is a reduce operation, which “reduces” an array of values down to a single value. Examples include the sum or product of the elements, the minimum, and the maximum. The following method takes an array and returns the sum of its elements:

```
public static double sum(double[] array) {
    double total = 0.0;
    for (int i = 0; i < array.length; i++) {
        total += array[i];
    }
    return total;
}
```

- Before the loop, we initialize total to zero. Each time through the loop, we update total by adding one element from the array. At the end of the loop, total contains the sum of the elements. A variable used this way is sometimes called an **accumulator**, because it “accumulates” the running total.

LECTURE 6

Random numbers

Non-deterministic Program

- Most computer programs do the same thing every time they run; programs like that are called **deterministic**. Usually determinism is a good thing, since we expect the same calculation to yield the same result. But for some applications, we want the computer to be unpredictable. Games are an obvious example, but there are many others, like scientific simulations.
- Making a program **nondeterministic** turns out to be hard, because it's impossible for a computer to generate truly random numbers. But there are algorithms that generate unpredictable sequences called **pseudorandom** numbers. For most applications, they are as good as random.

Random Class

- If you did Exercise 4, you have already seen **java.util.Random**, which generates pseudorandom numbers. The method nextInt takes an integer argument, n, and returns a random integer between 0 and n - 1 (inclusive).
- If you generate a long series of random numbers, every value should appear, at least approximately, the same number of times. One way to test this behavior of nextInt is to generate a large number of values, store them in an array, and count the number of times each value occurs.
- The following method creates an int array and fills it with random numbers between 0 and 99. The argument specifies the desired size of the array, and the return value is a reference to the new array.

Creation of A Random Array

```
public static int[] randomArray(int size) {
    Random random = new Random();
    int[] a = new int[size];
    for (int i = 0; i < a.length; i++) {
      a[i] = random.nextInt(100);
    }
    return a;
}
```

Array.toString()

- The following main method generates an array and displays it using printArray from Section 7.3. We could have used Arrays.toString, but we like seeing curly braces instead of square brackets.

```
public static void main(String[] args) {
    int[] array = randomArray(8);
    printArray(array);
}
```

- Each time you run the program, you should get different values. The output will look something like this:

```
{15, 62, 46, 74, 67, 52, 51, 10}
```

LECTURE 7

Building a histogram

In Range Test

- If these values were exam scores – and they would be pretty bad exam scores in that case – the teacher might present them to the class in the form of a histogram. In statistics, a histogram is a set of counters that keeps track of the number of times each value appears.
- For exam scores, we might have ten counters to keep track of how many students scored in the 90s, the 80s, etc. To do that, we can traverse the array and count the number of elements that fall in a given range.
- The following method takes an array and two integers. It returns the number of elements that fall in the range from low to high - 1.

Traversal for In-Range Test

```
public static int inRange(int[] a,int low, int high)
{
      int count = 0;
      for (int i = 0; i < a.length; i++) {
        if (a[i] >= low && a[i] < high) {
          count++;
        }
      }
      return count;
  }
```

inRange() function

- This pattern should look familiar: it is another reduce operation. Notice that low is included in the range (>=), but high is excluded (<). This design keeps us from counting any scores twice.
- Now we can count the number of scores in each grade range. We add the following code to our main method:

```
int[] scores = randomArray(30);
int a = inRange(scores, 90, 100);
int b = inRange(scores, 80, 90);
int c = inRange(scores, 70, 80);
int d = inRange(scores, 60, 70);
int f = inRange(scores, 0, 60);
```

Using inRange Method for Histogram

- This code is repetitive, but it is acceptable as long as the number of ranges is small. Suppose we wanted to keep track of the number of times each individual score appears. Then we would have to write 100 lines of code:

```
int count0 = inRange(scores, 0, 1);
int count1 = inRange(scores, 1, 2);
int count2 = inRange(scores, 2, 3);
...
int count99 = inRange(scores, 99, 100);
```

- What we need is a way to store 100 counters, preferably so we can use an index to access them. Wait a minute, that's exactly what an array does.

Use Loop for Abstraction

- The following fragment creates an array of 100 counters, one for each possible score. It loops through the scores and uses inRange to count how many times each score appears. Then it stores the results in the counts array:

```
int[] counts = new int[100];
for (int i = 0; i < counts.length; i++) {
    counts[i] = inRange(scores, i, i + 1);
}
```

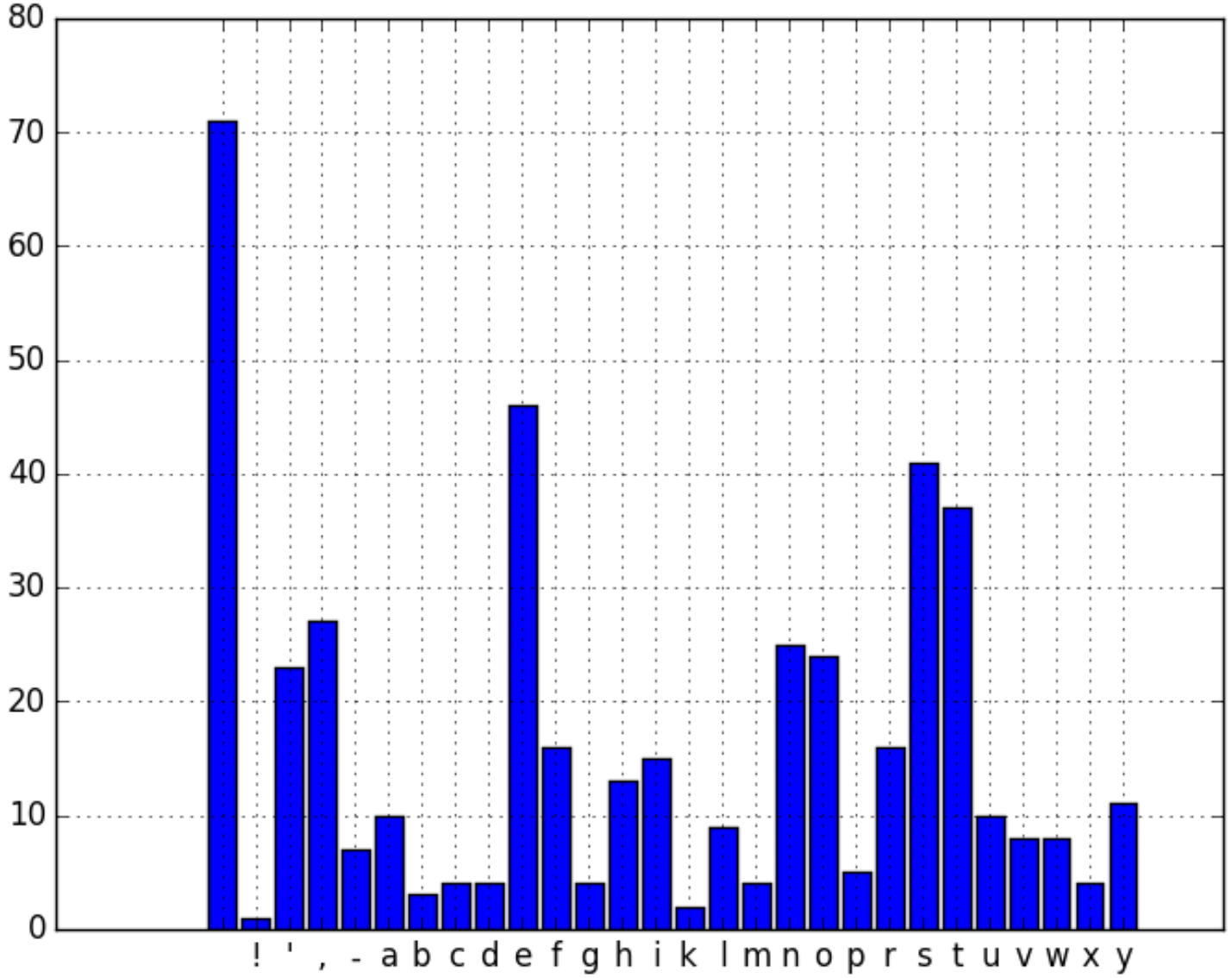
- Notice that we are using the loop variable i three times: as an index into the counts array, and in the last two arguments of inRange.

InRange Method

- The code works, but it is not as efficient as it could be. Every time the loop invokes inRange, it traverses the entire array. It would be better to make a single pass through the scores array.
- For each score, we already know which range it falls in – the score itself. We can use that value to increment the corresponding counter. This code traverses the array of scores only once to generate the histogram:

```
int[] counts = new int[100];
for (int i = 0; i < scores.length; i++) {
    int index = scores[i];
    counts[index]++;
}
```

- Each time through the loop, it selects one element from scores and uses it as an index to increment the corresponding element of counts. Because this code traverses the array of scores only once, it is much more efficient.


80
70
60
50
40
30
20
10
0
! ' , - a b c d e f g h i k l m n o p r s t u v w x y

LECTURE 8

The enhanced for loop

For-Each Loop

- Since traversing arrays is so common, Java provides an alternative syntax that makes the code more compact. Consider a for loop that displays the elements of an array on separate lines:

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

- We could rewrite the loop like this:

```
for (int value : values) {
    System.out.println(value);
}
```

For Each Element, Not For Each Index

- This statement is called an enhanced for loop, also known as the “for each” loop. You can read the code as, “for each value in values”. It’s conventional to use plural nouns for array variables and singular nouns for element variables.
- Notice how the single line for (int value : values) replaces the first two lines of the standard for loop. It hides the details of iterating each index of the array, and instead, focuses on the values themselves.
- Using the enhanced for loop, and removing the temporary variable, we can write the histogram code from the previous section more concisely:

```
int[] counts = new int[100];
for (int score : scores) {
    counts[score]++;
}
```

No Idea about the Index

- Enhanced for loops often make the code more readable, especially for accumulating values. But they are not helpful when you need to refer to the index, as in search operations.

```
for (double d : array) {
    if (d == target) {
      // array contains d,
      // but we don't know where
    }
}
```

LECTURE 9

Counting characters

Counting Loops

- We now return to the example from the beginning of the chapter and present a solution to Exercise 5 using arrays. Here is the problem again:

  A word is said to be a "**doubloon**" if every letter that appears in the word appears exactly twice. Write a method called isDoubloon that takes a string and checks whether it is a doubloon. To ignore case, invoke the toLowerCase method before checking.

- Based on the approach from Section 7.7, we will create an array of 26 integers to count how many times each letter appears. We convert the string to lowercase, so that we can treat 'A' and 'a' (for example) as the same latter.

Counting Characters - Histogram

```
int[] counts = new int[26];
String lower = s.toLowerCase();
```

- We can use a for loop to iterate each character in the string. To update the counts array, we need to compute the index that corresponds to each character. Fortunately, Java allows you to perform arithmetic on characters.

```
for (int i = 0; i < lower.length(); i++) {
    char letter = lower.charAt(i);
    int index = letter - 'a';
    counts[index]++;
}
```

Using for-each loop

- To simplify the code, it would be nice to use an enhanced for loop. The enhanced for loop does not work with strings directly, but you can convert any string to a character array and iterate that instead:

```
for (char letter : lower.toCharArray())
{
    int index = letter - 'a';
    counts[index]++;
}
```

Check for doubloon

- After counting all the characters in the lower string, we need one last for loop to determine whether each letter appears 0 or 2 times.

```
for (int count : counts) {
    if (count != 0 && count != 2) {
        return false;  // not a doubloon
    }
}
return true;  // is a doubloon
```

Discussion

- Like in Section 7.5, we can return immediately if the inner condition is true (which, in this example, means that the word is not a doubloon). If we make it all the way through the for loop, we know that all counts are 0 or 2.
- Pulling together the code fragments, and adding some comments and test cases, here is an entire program. It's amazing to think about how much you've learned in just seven chapters!

In-Class Demo Program

- Counting Characters

DOUBLOON.JAVA

Homework

Homework

- Textbook Exercises: All problems
- Project 8