Chapter 20

Arrays

Individual variables are classified as *scalars*. A scalar can assume exactly one value at a time. As we have seen, individual variables can be used to create some interesting and useful programs. Scalars, however, do have their limitations. Consider `AverageNumbers` (20.1) which averages five numbers entered by the user.

```
import java.util.Scanner;

public class AverageNumbers {
    public static void main(String[] args) {
        double n1, n2, n3, n4, n5;
        Scanner scan = new Scanner(System.in);
        System.out.print("Please enter five numbers: ");
        //  Allow the user to enter in the five values.
        n1 = scan.nextDouble();
        n2 = scan.nextDouble();
        n3 = scan.nextDouble();
        n4 = scan.nextDouble();
        n5 = scan.nextDouble();
        System.out.println("The average of " + n1 + ", " + n2
                           + ", " + n3 + ", " + n4 + ", " + n5
                           + " is " + (n1 + n2 + n3 + n4 + n5)/5);
    }
}
```

Listing 20.1: `AverageNumbers`—Average five numbers entered by the user

A sample run of `AverageNumbers` looks like:

```
Please enter five numbers: 9 3.5 0.2 100 15.3
The average of 9.0, 3.5, 0.2, 100.0, 15.3 is 25.6
```


AverageNumbers (20.1) conveniently displays the values the user entered and then computes and displays their average. Suppose the number of values to average must increase from five to 25. Twenty additional variables must be introduced, and the overall length of the program will necessarily grow. Averaging 1,000 numbers using this approach is impractical.

AverageNumbers2 (20.2) provides an alternative approach for averaging numbers.

```
import java.util.Scanner;

public class AverageNumbers2 {
    private static int NUMBER_OF_ENTRIES = 5;
    public static void main(String[] args) {
        double sum = 0.0;
        Scanner scan = new Scanner(System.in);
        System.out.print("Please enter " + NUMBER_OF_ENTRIES
                         + " numbers: ");
        //  Allow the user to enter in the five values.
        for ( int i = 0;  i < NUMBER_OF_ENTRIES;  i++ ) {
            sum += scan.nextDouble();
        }
        System.out.println("The average of the " + NUMBER_OF_ENTRIES
                           + " values is "
                           + sum/NUMBER_OF_ENTRIES);
    }
}
```

Listing 20.2: AverageNumbers2—Another program that averages five numbers entered by the user

It behaves slightly differently from the AverageNumbers program, as the following sample run using the same data shows:

```
Please enter five numbers: 9 3.5 0.2 100 15.3
The average of the five values is 25.6
```

AverageNumbers2 can be modified to average 25 values much more easily than the AverageNumbers that must use 25 separate variables (just change the constant NUMBER_OF_ENTRIES). In fact, the coding change to average 1,000 numbers is no more difficult. However, unlike AverageNumbers, AverageNumbers2 does not display the numbers entered. This is a significant difference; it may be necessary to retain all the values entered for various reasons:

- All the values can be redisplayed after entry so the user can visually verify their correctness.
- The values may need to be displayed in some creative way; for example, they may be placed in a graphical user interface component, like a visual grid (spreadsheet).
- The values entered may need to be processed in a different way after they are all entered; for example, we may wish to display just the values entered above a certain value (like greater than zero), but the limit is not determined until after all the numbers are entered.

In all of these situations we must retain the values of all the variables for future recall.

We need to combine the advantages of both of the above programs:

- `AverageNumbers`—the ability to retain individual values
- `AverageNumbers2`—the ability to dispense with creating individual variables to store all the individual values

An *array* captures both of these advantages in one convenient package. An array is a *nonscalar* variable. An array is a collection of values. An array has a name, and the values it contains are accessed via their position within the array.

20.1 Declaring and Creating Arrays

An array is a nonscalar variable. Like any other variable, an array can be a local, class, or instance variable, and it must be declared before it is used. Arrays can be declared in various ways:

- The general form of a simple, single array declaration is

$$type\texttt{[]}\ \ name\texttt{;}$$

  The square brackets denote that the variable is an array. Some examples of simple, single array declarations include:

```
int[] a;          //  a is an array of integers
double[] list;    //  list is an array of double-precision
                  //  floating point numbers
```

- Multiple arrays of the same type can be declared as

$$type\texttt{[]}\ \ name_1\texttt{,}\ \ name_2\texttt{,}\ \ \texttt{...,}\ \ name_n\texttt{;}$$

  The following declaration declares three arrays of characters and two Boolean arrays:

```
char[] a, letters, cList;
boolean[] answerVector, selections;
```

In each of the above cases, the number of elements is not determined until the array is created. In Java, an array is a special kind of *object*. An array variable is therefore an *object reference*. Declaring an array does not allocate space for its contents just like declaring an object reference does not automatically create the object it is to reference; the `new` operator must be used to create the array with the proper size. Given the declarations above, the following examples show how arrays can be created:

```
//  Declarations, from above
char[] a, letters, cList;
boolean[] answerVector, selections;
```



```
/*  . . .  */
//  Array allocations
a = new char[10];  //  a holds ten characters
letters = new char[scan.nextInt()];  //  Number of characters
                                       //  entered by user
cList = new char[100];  //  cList holds 100 characters
answerVector = new Boolean[numValues];  //  Size of answerVector
                                        //  depends on the value
                                        //  of a variable
```

An array has a declared type, and all elements stored in that array must be compatible with that type. Because of this, arrays are said to be *homogeneous* data structures.

Arrays may be initialized when they are declared; for example:

```
char[] a = new char[10];
```

This statement declares an array of characters and creates it with a capacity of ten elements. When an array is allocated, by default its contents are all *zero-like*. Zero-like means something different for different types, as Table 20.1 shows. Actually, these zero-like values are used as default values in two different situations:

Type	Value
`byte`, `short`, `int`, `char`, `long`, `float`, `double`	0
`boolean`	`false`
Object reference	`null`

Table 20.1: Zero-like values for various types

- they are assigned to elements of newly allocated arrays that are not otherwise initialized and
- they are assigned to class and instance variables that are not otherwise initialized.

It is possible to both create an array and initialize its contents simultaneously. The elements are provided in a comma separated list within curly braces:

$$type\texttt{[]}\ name\ \texttt{=}\ \{\ value_1\texttt{,}\ \ value_2\texttt{,}\ \ \ldots\texttt{,}\ \ value_3\ \}\texttt{;}$$

Examples include:

```
int[] a = { 20, 30, 40, 50 };
char[] cList = { 'a', 'b', 'c', 'd' };
```

This comma separated list of initial values is called an *initialization list* . No integer expression is used in the square brackets because the compiler can calculate the length of the initialization list. Notice that `new` is not used here even though an array object is being created on the heap; this special syntax is only possible when an array is declared with an initialization list. Variables can also be used with this special syntax:


```
int x = 10, y = 20, z = 30;
int[] a = { x, y, z };
```

Finally, it is possible to create an array with an initialization list sometime after it has been declared, as the following example shows:

```
int[] a;
/*  . . .  */
a = new int[] { 10, 20, 30 };
```

Notice that `new` operator must be used in this case.

Figure 20.1 illustrates how arrays are created.

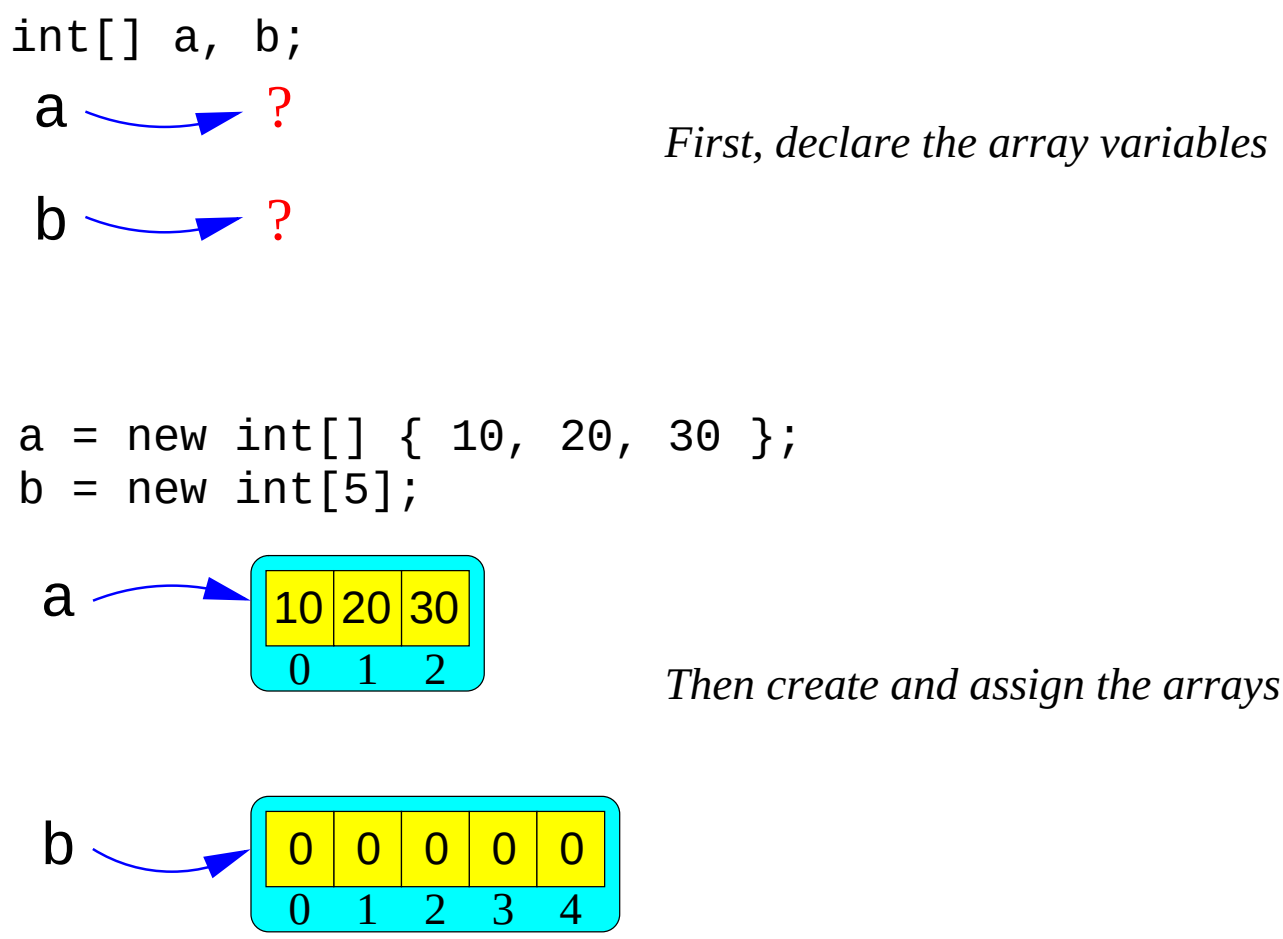

Figure 20.1: Creating arrays

20.2 Using Arrays

Once an array has been properly declared and created it can be used within a program. A programmer can use an array in one of two ways:

1. an element of the array can be used within a statement or

2. a reference to the whole array itself can be used within a statement.

In the next chapter we will see how an array as a whole can be used within a program; in this chapter we see how individual elements can be accessed.

Once an array has been created, its elements can be accessed with the `[]` operator. The general form of an array access expression is:

name [*expression*]

The expression within the square brackets must evaluate to an integer; some examples include

- an integer literal: `a[34]`
- an integer variable: `a[x]`
- an integer arithmetic expression: `a[x + 3]`
- an integer result of a method call that returns an `int`: `a[find(3)]`
- an access to an integer array: `a[b[3]]`

The square brackets and enclosed expression is called an *index* or *subscript*. The subscript terminology is borrowed from mathematicians who use subscripts to reference elements in a vector (for example, V_2 represents the second element in vector V). Unlike the convention used in mathematics, however, the first element in the array is at position zero, not one. The index indicates the distance from the beginning; thus, the very first element is at a distance of zero from the beginning of the array. The first element of array `a` is `a[0]`. If array `a` has been allocated to hold n elements, then the last element in `a` is `a[`$n-1$`]`. Figure 20.2 illustrates assigning an element in an array. The square bracket

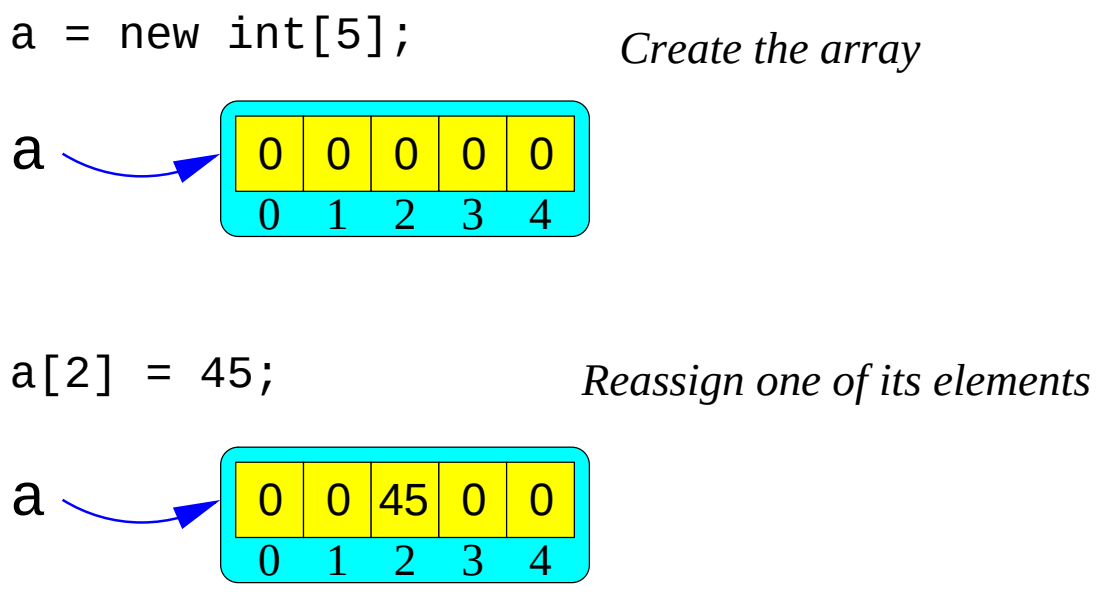

Figure 20.2: Assigning an element in an array

notation is one aspect that makes arrays special objects in Java. An object of any other class must use a method call

to achieve the same effect; for example, the `charAt()` method must be used to access individual characters within a `String` object.

The programmer must ensure that the index is within the bounds of the array. Since the index can consist of an arbitrary integer expression whose value cannot be determined until run time, the compiler cannot check for out-of-bound array accesses. A runtime error will occur if a program attempts an out-of-bounds array access. The following code fragments illustrate proper and improper array accesses:

```
double[] numbers = new double[10];  //  Declare and allocate array
numbers[0] = 5;   //  Put value 5 first
numbers[9] = 2.5; //  Put value 2.5 last
numbers[-1] = 5;  //  Runtime error; any negative index is illegal
numbers[10] = 5;  //  Runtime error; last valid position is 9
numbers[1.3] = 5; //  Compile-time error; nonintegral index illegal
```

Notice that the compiler does check that the type of the index is correct.

Loops are frequently used to access the values in an array. `ArrayAverage` (20.3) uses an array and a loop to achieve the generality of `AverageNumbers2` and the ability to retain all input for later redisplay:

```
import java.util.Scanner;

public class ArrayAverage {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        double[] numbers = new double[5];
        double sum = 0.0;
        System.out.print("Please enter five numbers: ");
        //  Allow the user to enter in the five values.
        for ( int i = 0;  i < 5;  i++ ) {
            numbers[i] = scan.nextDouble();
            sum += numbers[i];
        }
        System.out.print("The average of ");
        for ( int i = 0;  i < 4;  i++ ) {
            System.out.print(numbers[i] + ", ");
        }
        //  No comma following last element
        System.out.println(numbers[4] + " is " + sum/5);
    }
}
```

Listing 20.3: `ArrayAverage`—Use an array to average five numbers entered by the user

The output of `ArrayAverage` is identical to our original `AverageNumbers` (20.1) program:

```
Please enter five numbers:  9 3.5 0.2 100 15.3
The average of 9.0, 3.5, 0.2, 100.0, 15.3 is 25.6
```


but we can conveniently extend it to handle as many values as we wish.

Notice in `ArrayAverage` that the first loop collects all five input values from the user. The second loop only prints the first four because it also prints a trailing comma after each element; since no comma is to follow the last element, it is printed outside the loop.

`ArrayAverage` is less than perfect, however. In order to modify it to be able to handle 25 or 1,000 elements, no less than five lines of source code must be touched (six, if the comment is to agree with the code!). A better version is shown in `BetterArrayAverage` (20.4):

```
import java.util.Scanner;

public class BetterArrayAverage {
    private static final int SIZE = 5;
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        double[] numbers = new double[SIZE];
        double sum = 0.0;
        System.out.print("Please enter " + SIZE + " numbers: ");
        //  Allow the user to enter in the values.
        for ( int i = 0;  i < SIZE;  i++ ) {
            numbers[i] = scan.nextDouble();
            sum += numbers[i];
        }
        System.out.print("The average of ");
        for ( int i = 0;  i < SIZE - 1;  i++ ) {
            System.out.print(numbers[i] + ", ");
        }
        //  No comma following last element
        System.out.println(numbers[SIZE - 1] + " is " + sum/SIZE);
    }
}
```

Listing 20.4: `BetterArrayAverage`—An improved `ArrayAverage`

`BetterArrayAverage`'s output is only slightly different from that of `ArrayAverage`:

```
Please enter 5 numbers:  9 3.5 0.2 100 15.3
The average of 9.0, 3.5, 0.2, 100.0, 15.3 is 25.6
```

In `BetterArrayAverage`, only the definition of the constant `SIZE` must be changed to allow the program to average a different number of values. This centralization of the definition of the array's size eliminates redundancy and leads to a program that is more maintainable. When redundant information is scattered throughout a program, it is a common error to update some but not all of the information when a change is to be made. If all of the redundant information is not updated to agree, the inconsistencies result in errors within the program.

Sometimes it is necessary to determine the size of an array. All arrays have a `public final` attribute named `length` that holds the allocated size of the array. For example, the first `for` loop in `BetterArrayAverage` could be rewritten as


```
for ( int i = 0;  i < numbers.length;  i++ ) {
    numbers[i] = scan.nextDouble();
    sum += numbers[i];
}
```

Note that the last element in `numbers` is `numbers[numbers.length - 1]`. This feature is valuable when an array must be manipulated by code that did not allocate it (see § 21.1 for such an example).

A common idiom for visiting each element in an array `a` is

```
for ( int i = 0;  i < a.length;  i++ ) {
    //  do something with a[i]
}
```

If the elements of `a` are only being read and not modified, a variant `for` construct called the "for/each" loop can be used. If, for example, array `a` held `double`s, the above `for` loop could be rewritten

```
for ( double d : a ) {
    //  do something with d
}
```

You read the above statement as "for each `double d` in `a`, do something with `d`." Inside the body of the loop `d` represents a particular element of `a`: the first time through `d = a[0]`, the second time through `d = a[1]`, etc. The last time through the loop `d` represents `a[a.length - 1]`, the last element in `a`.

Where possible, the for/each construct should be used instead of the normal `for` construct. The for/each construct is simpler and has fewer pieces for the programmer to get wrong. The for/each loop cannot be used all the time, though:

- Since `d` is variable local to the for/each loop and `d` is a copy of an element in `a`, modifying `d` does not affect the contents of `a`. The statement

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

  can be readily transformed to the equivalent but simpler

  ```
  for ( double d : a ) {
      System.out.println(d);
  }
  ```

  but the statement

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

  cannot be transformed into


```
for ( double d : a ) {
    d = 0;
}
```

since the for/each version sets the copy of the array element to zero, not the array element itself.

- The for/each loop visits every element in the array. If only a subrange of the array is to be considered, the for/each loop is not appropriate and the standard `for` loop should be used.
- The for/each loop visits each element in order from front to back (lowest index to highest index). If array elements must be considered in reverse order or if not all elements are to be considered, the for/each loop cannot be used.
- The for/each statement does not reveal in its body the index of the element it is considering. If an algorithm must know the index of a particular element to accomplish its task, the traditional `for` loop should be used. The traditional `for` loop's control variable also represents inside the body of the loop the index of the current element.

20.3 Arrays of Objects

An array can store object references. Recall the simple class `Widget` (9.2) from Section 9.4. The following code

```
Widget[] widgetList = new Widget[10];
```

declares and creates an array of ten `Widget` references. It does not create the individual objects in the array. In fact, the elements of `widgetList` are all `null` references. The following code

```
Widget[] widgetList = new Widget[10];
for ( int w : widgetList ) {
    w.identify();
}
```

will compile fine but fail during execution with a *Null pointer exception*, since all the elements are `null`. It is an error to attempt to dereference a `null` pointer (in this case `null.identify()`). If `widgetList` is indeed to hold ten `Widget` objects, these objects must be created individually. Since the `Widget` constructor takes no arguments, this can easily be done in a loop:

```
Widget[] widgetList = new Widget[10];
for ( int i = 0;  i < widgetList.length;  i++ ) {
    widgetList[i] = new Widget();
}
```

This code ensures that `widgetList` holds ten viable `Widget` objects.

Because `Object` is the root of the Java class hierarchy, any reference type *is a* `Object`. This means any reference type (`String`, `TrafficLightModel`, `Random`, etc.) is assignment compatible with `Object`, as illustrated by the following legal code fragment:

```
Object obj = "Hello";  //  OK, since a string IS an object
```


An array of `Object`s, therefore, can hold references of *any* type:

```
Object[] obj = { "Hello", new Random(),
                 new TrafficLightModel(TrafficLightModel.STOP),
                 new Rational(1, 2) };
```

Furthermore, since all primitive types have associated wrapper classes, and since primitive values are autoboxed when required (see § 13.4), the following code is legal as well:

```
Object[] obj = { "Hello", new Random(), 12,
                 new Rational(1, 2), 3.14, "End" };
```

We said that arrays hold *homogeneous* types; for example, an array of `int`s can only hold `int` values. This is true, but because anything is assignment compatible with `Object`, an `Object` array can hold a diverse collection of *heterogenous* values. Of course the values are heterogenous with respect to each other, but they are homogeneous in the fact that they are all `Object`s. Their behavior within the array is differentiated by polymorphism:

```
//  obj is the array of Objects from above
for (Object elem : obj) {
    System.out.println(elem);
}
```

The `toString()` method in each element determines how each element is displayed.

Since an array is itself an object reference, arrays of arrays are possible. Section 20.4 explores arrays of arrays, commonly called multidimensional arrays.

20.4 Multidimensional Arrays

Java supports arrays containing any type—even arrays of arrays. The statement

```
int[][] a;
```

declares `a` to be an array of integer arrays. It is convenient to visualize this array of arrays as a two-dimensional (2D) structure like a table. A 2D (or higher dimension) array is also called a *matrix*. Figure 20.3 shows a picture of the array created by the following sequence of code:

```
int[][] a;
a = new int[2][3];  //  Creates a 2D array filled with zeros
a[0][0] =  5;
a[0][1] = 19;
a[0][2] =  3;
a[1][0] = 22;
a[1][1] = -8;
a[1][2] = 10;
```

The two-dimensional array `a` is said to be a 2×3 array, meaning it has two rows and three columns (as shown in Figure 20.3). The first index signifies the row and the second index denotes the column of the element within the array.

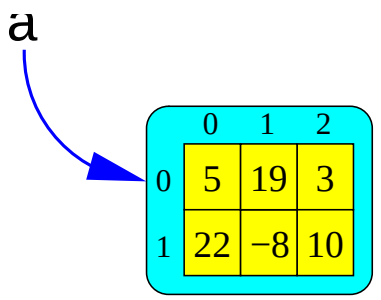

Figure 20.3: A 2×3 two-dimensional array

Using a syntax similar to the initialization lists of one-dimensional arrays, `a` above could have been declared and created as:

```
int[][] a;
a = new int[][] { new int[] {  5, 19,  3 },
                  new int[] { 22, -8, 10 } };
```

or even more compactly declared and initialized as:

```
int[][] a = { {  5, 19,  3 },
              { 22, -8, 10 } };
```

Each element of a 2D array is a 1D array. Thus, if `a` is a 2D array and `i` is an integer, then the expression

```
a[i]
```

is a 1D array. To obtain the scalar element at position `j` of this array `a[i]`, double indexing is required:

```
a[i][j]   //  The element at row i, column j
```

The following statement assigns the value 10 to the element at row 1, column 2:

```
a[1][2] = 10;
```

Since 2D arrays in Java are really arrays of arrays and are not tables, rows are not required to contain the same number of elements. For example, the following statement

```
int[][] a = { {  5, 19,  3 },
              { 22, -8, 10, 14, 0, -2, 8 },
              { 20, -8 },
              { 14, 15,  0,  4, 4 } };
```

creates what is known as a *jagged table* with unequal row sizes. Figure 20.4 illustrates this jagged table.

Since a 2D array is an array of 1D arrays and each of these 1D arrays are treated as rows, the `length` attribute of a 2D array specifies the number of rows. Each element of the 2D array represents a row which is itself an array with its own `length` attribute. Thus, given the declaration:

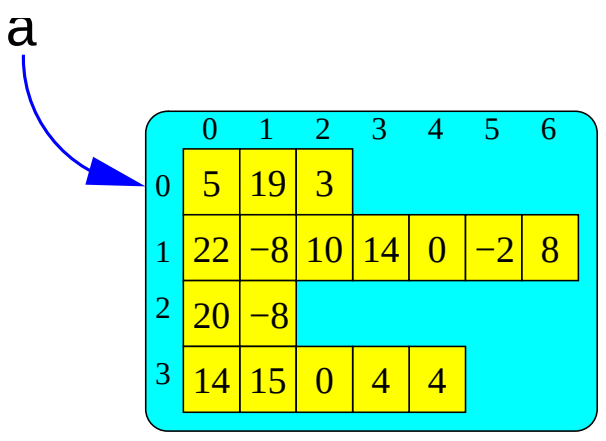

Figure 20.4: A “jagged table” where rows have unequal lengths

```
int[][] a;
```

- `a.length` represents the number of rows in array `a`
- `a[`*i*`]` represents row *i*, itself an array
- `a[`*i*`].length` is the number of elements in row *i*
- `a[`*i*`][`*j*`]` represents an integer element at row *i*, column *j*

The following method displays the contents of a 2D integer array:

```
public static void print2Darray(int[][] a) {
    for ( int row = 0;  row < a.length;  row++ ) {
        for ( int col = 0;  col < a[row].length;  col++ ) {
            System.out.print(a[row][col] + " ");
        }
        System.out.println();  //  Newline for next row
    }
}
```

The outer loop iterates over the rows (`a.length` is the number of rows); the inner loop iterates over the elements (columns) within each row (`a[row].length` is the number of elements in row `row`).

Arrays with dimensions higher than two can be represented. A 3D array is simply an array of 2D arrays, a 4D array is an array of 3D arrays, etc. For example, the statement

```
matrix[x][y][z][t] = 1.0034;
```

assigns 1.0034 to an element in a 4D array of `double`s. In practice, arrays with more than two dimensions are rare, but advanced scientific and engineering applications sometimes require higher-dimensional arrays.

20.5 Summary

- Add summary items here.

20.6 Exercises

