Chapter 8. Arrays and Files

In the preceding chapters, we have used variables to store single values of a given type. It is sometimes convenient to store multiple values of a given type in a single collection variable. Programming languages use arrays for this purpose. It is also convenient to store such values in files rather than by hard-coding them in the program itself or by expecting the user to enter them manually. Languages use files for this purpose. This chapter introduces the use of arrays and files in Java and Processing.

As in previous chapters, the running example is implemented in Processing but the remainder of the examples on arrays, array topics and multi-dimensional array can work in either Processing or Java. The material on files is Processing-specific; Java files are treated in a later chapter.

8.1. Example: Visualizing Data

Computers are powerful tools both for collecting and storing large amounts of *data* and for analyzing and presenting the patterns and trends in that data. These patterns and trends are commonly called *information*, and the computer is commonly used as a tool for deriving information from data. For example, computers have been used to collect and store thousands of statistics on human life and health for the United Nations, millions of customer records for multinational corporations, and billions of data points for the human genome project. Computers have also been used to mine useful information from these data sets. Processing provides all of these capabilities, with a particular emphasis on *data visualization*, whose goal is to present data in such as way as to allow humans to see the informational "big picture" that is so easily lost in the volumes of raw data.

Note that data representation and visualization are not easy tasks. Collecting and managing large data sets is challenging because of the myriad ways in which the data can be corrupted or lost. Processing large data sets requires considerable computing power and careful programming. Presenting data accurately requires careful extraction of data abstractions that are faithful to the original data. The entire field of *information systems*, a sub-field of computing, has arisen to address these issues.

In this chapter, our vision is to build an application that can display an appropriate set of data as a bar chart such as the one shown in the rough sketch in Figure 8-1. This is a standard bar chart in which each labeled bar represents a single data value and we'd like to add some aggregate statistics at the bottom. Bar charts such as this one allow the human visual system to perceive the relative values in this data set. Our goal is to display the average life expectancy in years of a newborn child in the five permanent members of the UN Security Council for the year 2007.

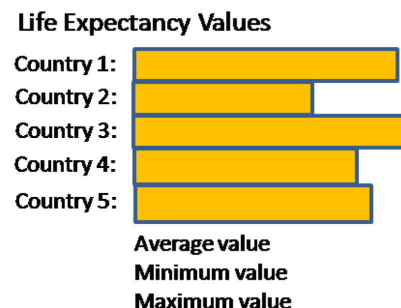


Figure 8-1. A bar chart showing created from a list of data values

Building a visualization such as this one requires that the application be able to:

- Represent the data, including the life expectancy values and the corresponding country names;
- Analyze the data and derive aggregate statistics (e.g., average, minimum and maximum values);
- Store the data permanently;
- Present the data in a visual bar chart.

We can achieve the last element of the vision, presenting the data as text and bars of appropriate sizes, using techniques discussed in the previous chapters. This chapter focuses on the first three elements. We will use arrays to represent the data, array processing techniques to analyze the data, and files to store the data permanently.

8.2. Arrays

The first element of the chapter example vision is to represent the five data values shown in Figure 8-1. In previous chapters, we would do this using five separate variables:

```
float expChina = 72.961,
      expFrance = 80.657,
      expRussia = 65.475,
      expUK = 79.425,
      expUSA = 78.242;
```

This approach could work, but consider the problem of computing the average of these data values. This would require the use of the following expression:

```
(expChina + expFrance + expRussia + expUK + expUSA) / 5
```

Now, consider the fact that the International Standards Organization officially recognizes over 200 countries. This means that working with data for all the countries would require over 200 separate `float` variables and expressions with separate operands to match.

As an alternative to the simple variable, which stores exactly one value, Java provides a data structure that stores multiple values of the same type. We have already seen an example of this sort of structure; Java represents variables of type `String` as lists of `char` values that can be accessed using an *index.* For example, if `aString` is a variable of type `String`, then `aString.charAt(i)` will return the `char` value in `aString` with index `i`. This section describes how to declare, initialize and work with indexed structures.

8.2.1. Declaring and Initializing Arrays

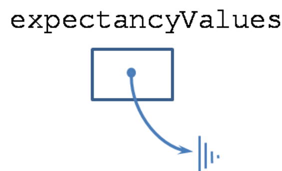
Java represents indexed data structures using *arrays*. Arrays are objects, which means that they must be accessed via handles. To define an array, we must *declare* an array handle and *initialize* the value referred to by that handle. To declare an array handle, we use the following pattern:

```
type[] name;
```

Here, *`type`* specifies the kind of values the array can store (e.g., float), the brackets (`[]`) indicate that an array is being defined and *`name`* is the handle through which the array can be accessed. For example, to declare an array of `float` values, we use the following code:

```
float[] expectancyValues;
```

This declaration tells Java that the `expectancyValues` handle references an array of floats. The array can be of any size. The data structure produced by this declaration can be viewed as follows:



Here, the `expectancyValues` handle is ready to reference an array of float values but currently references only a `null` value, denoted here by an electrical ground symbol. Before this handle can be used, we must replace this `null` value by creating an array.

Array Creation and Initialization using `new`

The typical way to create a new array is to use the `new` operation; this is the customary approach for reference types. The pattern for this creation is as follows.

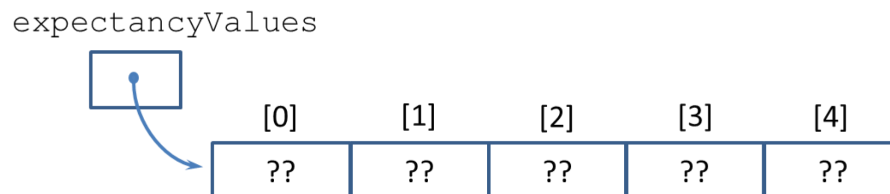
```
new type[size]
```

Here, type specifies the *`type`* of values the array can store and *`size`* represents the number of those values that must be stored. Java automatically allocates a sufficient amount of contiguous memory for the specified number of values of the specified type.

For example, the following code allocates a block of memory large enough to represent five float values and initializes expectancyValues as a handle for that memory.

```
final int COUNTRY_COUNT = 5;
float[] expectancyValues = new float[COUNTRY_COUNT];
```

We can create an array of any size, but once created, the size remains fixed. This data structure can be visualized as follows:



Here, `expectancyValues` is a handle that points to a set of five adjacent values. Each value, known as an array *element,* is of type `float` and may change during the execution of program as is the case with variables. Compilers often initialize **`float`** values to 0.0, but it is unwise for a program to assume this without explicitly initializing the values itself (as described in the next section). Java initializes this

data structure by allocating a fixed amount of adjacent memory locations appropriate for representing the number and type of the elements. Once initialized, the length of the array cannot be modified.

Array Indexes

Each array value has an assigned index running from 0 to 4, shown in the figure using square braces. A program can access an individual array element using the *subscript* operator (`[]`), which requires the array's name and the item's index value. The pattern for using this operator to access the element of the array *anArray* of index *i* is shown here:

```
anArray[i]
```

In Java, array indexing uses a zero-based scheme, which means that the first item in the `expectancyValues` array can be accessed using the expression `expectancyValues[0]`, the second value using the expression `expectancyValues[1]`, and so forth. These subscript expressions are known as *indexed variables* because a program can use them as it uses any other variable. For example, a program can set the value of the first expectancy variable using this assignment statement.

```
expectancyValues[0] = 72.961;
```

When the program is running, Java's subscript operation tests the value of the index and throws an error if the index value is out of bounds. For example, the evaluating the expressions `expectancyValues[-1]` or `expectancyValues[6]` will throw errors.

ArrayLength

The number of elements in an array is known as the *length* of the array and can be accessed using the array `length` property.[1] For example, the length of the `expectancyValues` array can be accessed using `expectancyValues.length`, which returns integer value 5, and the last element of the array can be accessed using `expectancyValues[expectancyValues.length – 1]`.

Array Initializers

Java supports a way to initialize array values using array *initializers*. The following code initializes an array with the values shown in Figure 8-1.

```
float[] expectancyValues = {72.961, 80.657, 65.475, 79.425, 78.242};
```

This code initializes the values in the array to the literal `float` values specified in the braces (`{}`). In this case, Java allocates the size of the array data structure to fit the number of values found in the array initializer expression. This array initializer cannot be used as an array literal in other contexts; it must be used to initialize the array as shown here.

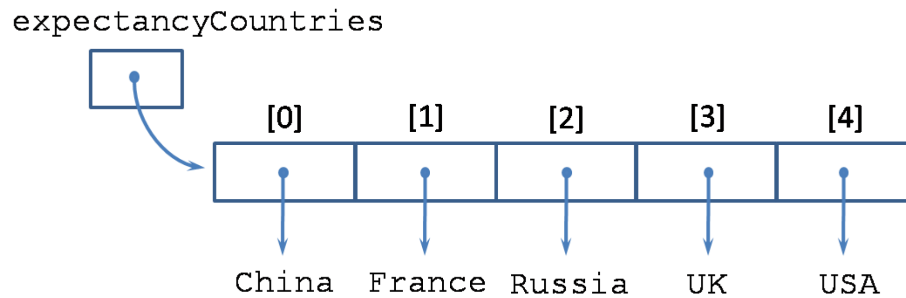
Java does not require arrays to store only values of primitive types. Arrays can store reference types as well. This statement defines an array of string objects:

```
String[] expectancyCountries =
        {"China", "France", "Russia", "UK", "USA"};
```

---

[1] In Java, a program accesses an array's length using the `length` property, e.g., `anArray.length`, whereas it accesses a string's length using the `length()` method, e.g., `aString.length()`.

This data structure can be visualized as follows:



Here the array elements are not primitive values, but handles for `String` reference objects.

Array definitions in Java have the following general pattern:

Array Definition Pattern

```
ElementType[] arrayName;
```

or

```
ElementType[] arrayName = new ElementType[length];
```

or

```
ElementType[] arrayName = arrayInitializer;
```

- *`ElementType`* is any type (including an array type);
- *`arrayName`* is the handle for the array object being defined – if there is no assignment clause in the statement, the handle value is set to `null`;
- *`length`* is an expression specifying the number of elements in the array;
- *`arrayInitializer`* is the list of literal values of type *`ElementType`*, enclosed in curly braces (`{ }`).

8.2.2. Working with Arrays

Because Java implements arrays as fixed length, indexed structures, the counting `for` loop provides an effective way to work with array elements. For example, the following code prints the names of the five countries represented by `expectancyCountries`:

Code:

```
for (int i = 0; i < expectancyCountries.length; i++) {
  println(i + ": " + expectancyCountries[i]);
}
```

Output:

```
0: China
1: France
2: Russia
3: UK
4: USA
```

This code loops through each index value of the `expectancyCountries` array, from `0` to `expectancyCountries.length - 1`, printing out the index value `i` and the value of the array element at that index value. Note that this code works regardless of the length of the `expectancyCountries` array.

Arrays and Methods

Programs can pass arrays as parameters and produce them as return values. For example, the following method receives an array from its calling program and returns the average of the values in the array:

```
float computeAverage(float[] values) {

  // Verify that the array actually has values first.
  if ((values == null) || (values.length <= 0)) {
    return 0.0;
  }

  // Compute and return the average.
  float sum = 0.0;
  for (int i = 0; i < values.length; i++) {
    sum += values[i];
  }
  return sum / values.length;
}
```

This method specifies a single parameter of type `float[]`; an array of any size can be passed to this method via such a parameter. The method first checks the parameter and returns 0.0 if the array handle is `null` or if the array is empty. This prevents null pointer and division by zero errors. If the **`values`** array passes these tests, then the method computes and returns the average of the float values.

The method uses a common algorithmic pattern called the *accumulator pattern*, in which the `sum` variable is used to accumulate the total value of the array entries. We will use this pattern frequently when working with arrays.

Methods can also return array objects. For example, the following method constructs and returns an array of a specified number of 0.0 values:

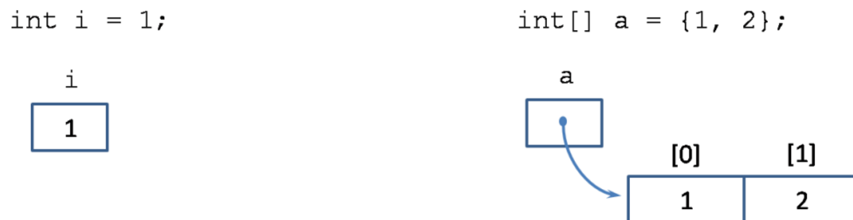
```
float[] constructZeroedArray(int arrayLength) {
  if (arrayLength < 0) {
    arrayLength = 0;
  }
```

```
float[] result = new float[arrayLength];
  for (int i = 0; i < arrayLength; i++) {
    result[i] = 0.0;
  }
  return result;
}
```

This method receives an integer representing the length of the desired array, verifies that it is at least 0, constructs the array, fills the array with values of 0.0, and finally returns the array. Note that Java and some other languages often initialize numeric array values to 0, but as discussed in the previous section, it generally not a good idea for a program to assume this.

Reference Types as Parameters

In Chapter 3 we discussed the distinction between primitive types and reference types, where primitive types store simple values and reference types store references, or pointers, to values. Arrays are reference types. The following diagram illustrates the difference:



On the left, we declare an integer `i`, whose value is the primitive integer value 1 as shown; on the right, we declare an integer array `a` whose value is a reference to the two-valued array as shown.

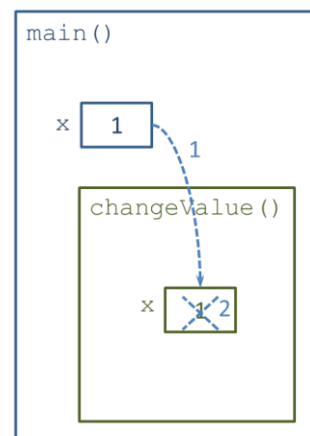
This distinction is important when using arrays and other reference types as parameters. Consider the following code, which initializes an integer variable `i` to the value 1 and passes that primitive value as an argument to the `changeValue()` method.

Code:

```
public static void main(String[] args) {
	int x = 1;
	changeValue(x);
	System.out.println("In main(), x == " + x);
}

public static void changeValue(int x) {
	x = 2;
	System.out.println("In changeValue(), x == " + x);
}
```

Output:

```
In changeValue(), x == 2
In main(), x == 1
```



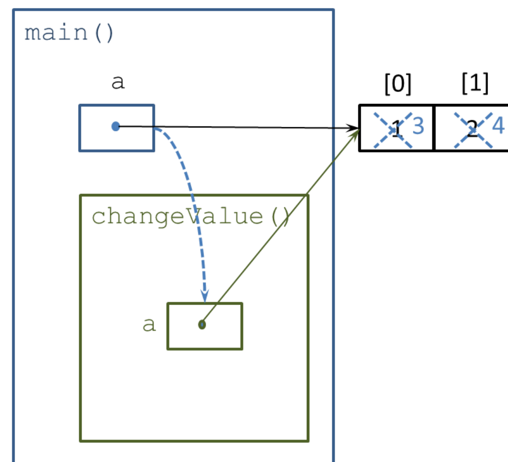
This code behaves as we would expect given our discussion of parameter passage in Chapter 4:

1. When `main()` calls `changeValue()`, it computes the value of its argument expression, `x`, and copies that value into `changeValue()`'s formal parameter, also called `x`. Note that there are two variables named `x`, one in `main()`'s scope and one in the `changeValue()`'s scope;
2. `changeValue()` then changes the value of its copy of `x` to 2 and prints this new value;
3. Finally, Java returns control to `main()`, which prints out the value of its copy of `x` (still 1).

In this parameter passage technique, called *pass-by-value*, the value of the argument is passed to the parameter. Java passes all of its parameters by value. However, because an array is a reference object, the pass-by-value technique leads to potentially unexpected results. Consider the following code, which initializes an integer array variable `a` to the array initializer value `{1, 2}` and passes that reference value as an argument to the `changeArray()` method:

Code:

```
public static void main(String[] args) {
      int[] a = { 1, 1 };
      changeArray(a);
      System.out.println("In main(): " + "{" +
            a[0] + ", " + a[1] + "}");
}

public static void changeArray(int[] a) {
      a[0] = 3;
      a[1] = 4;
      System.out.println("In changeArray(): " +
            "{" + a[0] + ", " + a[1] + "}");
}
```



Output:

```
In changeArray(): {3, 4}
In main(): {3, 4}
```

Note that the output of this code is different from the example given earlier. Here, `changeArray()` changes the values in the array permanently, which is why both calls to `println()` print the new values (3, 4). This code behaves as follows:

1. When `main()` calls `changeArray()`, it computes the value of its argument expression, `a`, and copies that reference value into `changeArray()`'s formal parameter, also called `a`. Java is still passing the array reference by value, but you can see in the diagram that the actual storage for the array values, referenced by a, is not copied;
2. `changeArray()` then changes the value of its copy of `a` to 3,4 and prints this new value;
3. Finally, Java returns control to `main()`, which prints out the value referenced by its copy of `a` (now 3, 4).

So while this is still pass-by-value behavior, the nature of the reference value being passed allows the original value of the argument to be accessed and changed by reference.

Though strings are reference types, implemented as arrays of characters, Java provides some features that allow programmers to work with them as primitive types. For example, one can initialize a String value by saying **`String myString = "a string value"`** rather than using **`new`** and Strings are passed by value to Java methods rather than by reference.

8.2.3. Predefined Array Operations

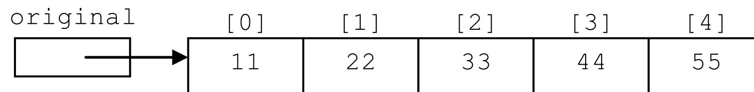
Java provides a few operations that can be used with arrays, including the assignment operator (=), the **`clone()`** method, and the **`equals()`** method. We will now take a closer look at each of these operations.

Array Assignment

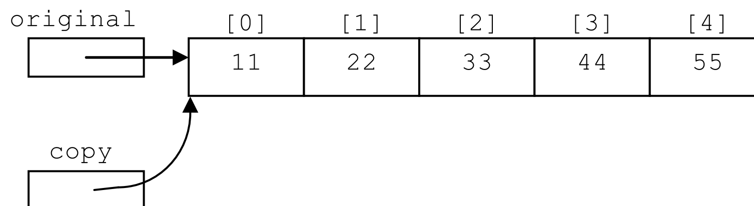
Java permits assignment expressions using array operands. For example, suppose that a program contains the following statements:

```
int [] original = {11, 22, 33, 44, 55};
int [] copy;
copy = original;
```

Although one might expect the third statement to define **`copy`** as a distinct copy of original, this is not what happens. The reason is that **`original`** and **`copy`** are both handles for array objects and are not arrays themselves. To see the difficulty, suppose we visualize the effect of the first statement as follows:



Once **`copy`** has been declared, then the third (assignment) statement simply copies the *address* from the handle **`original`** into the handle **`copy`**, which we might picture as



Thus, **`original`** and **`copy`** both refer to the same array object.

If the programmer was relying on the two handles referring to distinct arrays, then this represents a logic error – any change to the array referred to by original will simultaneously change the array referred to by copy. Therefore, the assignment operator should never be used to make a copy of an array.

Array Cloning

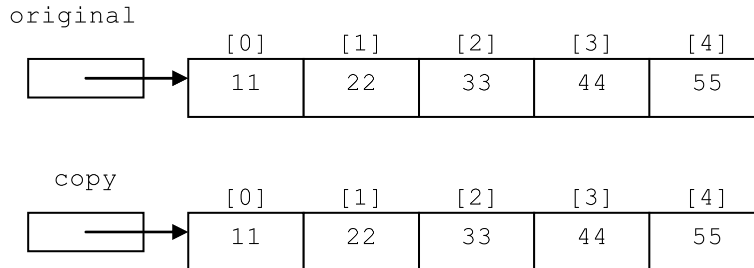
There are times when we need to create separate copies of Java objects. To support this, most predefined Java objects — arrays, in particular — have a **`clone()`** method that tells an object to make a copy of itself and return the address of the copy. To illustrate, if **`copy`** and **`original`** are handles for integer arrays as before,

```
int [] original = {11, 22, 33, 44, 55};
int [] copy;
```

and we want copy to be a distinct copy of original, we can write:

```
copy = original.clone();
```

The **clone()** method makes a distinct copy of the array original We can picture the result as follows:

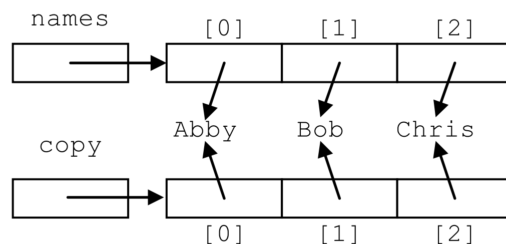


The **clone()** method can thus be used to make a distinct copy of an array.

It is important to note, however, that, for the sake of efficiency, **clone()** makes a *simple copy* of the object's memory. For arrays of primitive types such as **original**, this produces a completely distinct copy but *not* for arrays of reference types. To illustrate, consider the following code segment, which manipulates an array of **StringBuffer** objects:

```
StringBuffer[] names = { new StringBuffer("Abby"),
                         new StringBuffer("Bob"),
                         new StringBuffer("Chris") };
StringBuffer[] copy = names.clone();
```

In this example, we can picture the objects produced by these statements as follows:

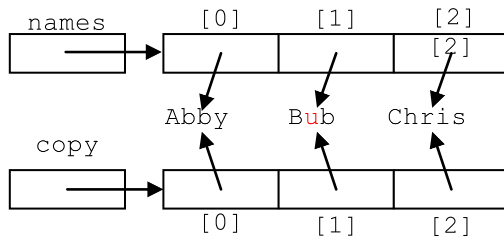


Here, the **clone()** method does makes a copy of the array **names**, but it is not a completely distinct copy. The reason is that **names** is an array of StringBuffer values, meaning its elements are StringBuffer handles. **StringBuffer** is similar to **String** except that where modifications to **String** objects result in the creation of a completely new string object, modifications of **StringBuffer** objects modify the existing **StringBuffer** object. When **names** is cloned, it makes a copy of itself by a simple copy of its memory. This creates a second array whose elements are copies of its elements, and since those elements are **String** handles containing addresses, the **String** handles in this copy contain the same addresses. Put differently, the elements of **names** and the elements of copy are different handles for the same sequence of values. Because it copies handles without copying the objects to which they refer, the **clone()** method's operation is sometimes referred to as a *shallow* copy operation.

In some situations, shallow copying can lead to a problem. The most common problem occurs if we change the objects to which the handles in a shallow copy refer. For example, if we use **names** to change the 'o' in "Bob" to 'u',

```
names[1].setCharAt(1, 'u');
```

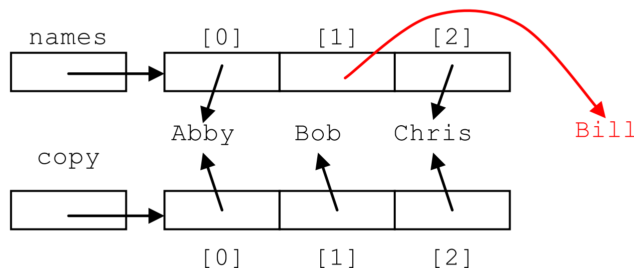
this change simultaneously affects the **StringBuffer** to which both **names[1]** and **copy[1]** refer:



To avoid such problems, we can write our own *deep copying* method. To illustrate, here is such a method for an array of **StringBuffer** objects:

```
public static StringBuffer [] deepCopy(StringBuffer [] original) {
  StringBuffer [] result = new StringBuffer[original.length];

  for (int i = 0; i < original.length; i++)
    result[i] = original[i].clone();

  return result;
}
```

There are many situations in which the **clone()** method's shallow copy is perfectly adequate, however. For example, if we assign **names[1]** the value **"Bill"**, **copy[1]** will still refer to "Bob":



Array Equality

Java's **Object** class defines an **equals()** message that can be sent to an array object:

```
if (a1.equals(a2) ) // ...
```

Unfortunately, this method simply compares the addresses in the *handles* a1 and a2. If they refer to the same object, then it returns true; otherwise it returns false. To actually compare the *elements* of two arrays, we must write our own method. To illustrate, the following class method **equals()** can be used to compare the elements of two arrays of **double** values, **array1** and **array2**:

```
public static boolean equals(double[] array1, double[] array2) {
  if (array1.length == array2.length) {
    for (int i = 0; i < array1.length; i++) {
      if (array1[i] != array2[i]) {
        return false;
      }
    }
    return true;
  } else {
    return false;
  }
}
```

The method first checks whether the lengths of the two arrays are the same; if not it returns false. Otherwise, it iterates through the index values, comparing the two arrays an element at a time. The method returns false if a mismatch is found, but returns true if it makes it through all index values without finding a mismatch.

A method to determine if two arrays of **`String`** values are equal uses the **`equals()`** method in place of **`==`** to compare the array elements. This is because the elements of the array are handles for **`String`** values, and the **`String`** class supplies its own definition of **`equals()`** to properly compare **`String`** values.

```
public static boolean equals(String[] array1, String[] array2) {
  if (array1.length == array2.length) {
    for (int i = 0; i < array1.length; i++) {
      if (!(array1[i].equals(array2[i]))) {
        return false;
      }
    }
    return true;
  } else {
    return false;
  }
}
```

A similar method can be used to compare arrays whose elements are of other reference types that define **`equals()`** properly.

8.2.4. Example Revisited

To build the chapter example, we must represent the name and life expectancy value for each country in our data set. Given that country names are strings and life expectancy values are floats, we will need to use two separate arrays to represent this data. To do this, we'll use the two arrays presented in this section: `expectancyCountries,` a string array, and `expectancyValues`, a float array. To keep track of the correspondence between the name of the county and the expectancy value for that country, we will make sure that the indexes of corresponding data match up properly. For example, `expectancyCountries[0]` should contain the name of the country whose expectancy value is stored in `expectancyValues[0]`. In this way, the index 0 co-indexes the name and value for one country.

We must also be able to print our data in a consistent manner. Our ultimate goal is to produce a bar chart such as the one shown in our original sketch shown in Figure 8-1. In this iteration, we'll satisfy ourselves by simply printing the names and values for each country without the bars. We'll also include the aggregate statistics. To achieve this preliminary goal, we can use the following algorithm.

Given:

- `expectancyCountries` is declared as an array of strings and is initialized with a list of country names. The index values correspond with the values of the expectancy value array.
- `expectancyValues` is declared as an array of floats and initialized with a list of expectancy values. The index values correspond with the values of the country name array.

Algorithm:

1. Print the table header.
2. Set `sum` = 0;
3. Set `maximum` = to a really small number.
4. Set `minimum` = to a really big number.
5. Repeat for a counter `i` ranging from 0 to the number of countries:
    a. Print a row in the table, with the current country name and expectancy value (that is, the `i`th country name and value).
    b. Set `sum` = `sum` + the current expectancy value.
    c. If the current expectancy value > `maximum`:
        i. Set `maximum` = the current expectancy value.
    d. If the current expectancy value < `minimum`:
        i. Set `minimum` = the current expectancy value.
6. Print the summary statistics: average (i.e., `sum` / number of countries), `maximum` and `minimum`.

This algorithm combines four basic tasks all in one loop. The main task is that of printing the table, which the algorithm does using a counting for loop that goes through the countries one at a time, printing one table row on each pass (steps 5 and 5.a). Each time through the loop, the algorithm refers to the "current" country name or expectancy value; this refers to the `i`th name or value in the respective arrays.

The loop is also computing statistics as it goes through. It computes the average expectancy value using the same algorithm shown in the section above (see the `computeAverage()` method). It's also searching for the maximum and the minimum life expectancy values. It does this by maintaining a maximum (and minimum) value "seen so far". Each time through the loop, it updates these values based on whether the current value is larger (or smaller) than the current value seen so far. All three of these accumulator algorithms assume that their accumulators have been initialized properly before the loop starts. The sum accumulator must be initialized to 0, which ensures that sum accumulated by the loop is accurate. The maximum accumulator must be set to some really small number, which ensures that the current value seen the first time through the loop will always be larger than the maximum value seen so far. The computation of the minimum value is handled similarly.

The following code implements this algorithm.

```
/**
 * ExpectancyTable displays a textual table of part of GapMinder's
 * life-expectancy data for 2007 (see http://www.gapminder.org/).
 *
 * @author kvlinden, snelesen
 * @version Fall, 2011
 */
final String[] expectancyCountries = { "China", "France", "Russia", "UK",
"USA" };
final float[] expectancyValues = { 72.961, 80.657, 65.475, 79.425, 78.242 };
final String year = "2007", source = "GapMinder.com, 2009";

void setup(){
   // Print the table header.
   println("Average Life Expectancy in Years (" + year + ")");

   // Initialize the aggregator values.
   float sum = 0.0, maximum = Float.MIN_VALUE, minimum = Float.MAX_VALUE;

   for (int i = 0; i < expectancyCountries.length; i++) {
      // Print the next table row.
      print(expectancyCountries[i] + ": " + expectancyValues[i] + "\n");

      // Accumulate the sum of the expectancy values.
      sum += expectancyValues[i];

      // Update the maximum value seen so far.
      if (expectancyValues[i] > maximum) {
        maximum = expectancyValues[i];
      }

      // Update the minimum value seen so far.
      if (expectancyValues[i] < minimum) {
        minimum = expectancyValues[i];
      }
   }

   // Print the aggregate statistics.
   println("Average: " + sum / expectancyCountries.length);
   println("Maximum Value: " + maximum);
   println("Minimum Value: " + minimum);
   println("Data Source: " + source);
}
```

This program prints the following simple table in the text output panel.

```
Average Life Expectancy in Years (2007)
China: 72.961
France: 80.657
Russia: 65.475
UK: 79.425
USA: 78.242
Average: 75.352005
Maximum Value: 80.657
Minimum Value: 65.475
Data Source: GapMinder.com, 2009
```

This code represents its raw data using two arrays, `expectancyCountries` and `expectancyValues`, and initializes them using array initializers. It uses the `Float` library constants `Float.MIN_VALUE` and `Float.MAX_VALUE` in the maximum and minimum value computation described above; these values are set automatically to represent the smallest (largest) `float` values.

8.3. Array Topics

There are a number of important problems in computing that can be addressed using arrays. This section introduces one of these topics: search.

8.3.1. Searching

One important computational problem is *searching* a collection of data for a specified item and retrieving some information associated with that item. For example, one searches a telephone directory for a specific name in order to retrieve the phone number listed with that name. We consider two kinds of searches, linear search and binary search.

Linear Search

A *linear search* begins with the first item in a list and searches sequentially until either the desired item is found or the end of the list is reached. The following algorithm specifies a method that uses this approach to search a list of *n* elements stored in an array, *list*[0], *list*[1], . . ., *list*[*n* – 1] for *value*. It returns the location of *value* if the search is successful, or the value -1 otherwise.

Linear Search Algorithm:

1. **Receive** a non-null *list* of values and a target *value*.
2. **Loop** for each element in *list*
    a. **If** *value* equals *list*[i] **then**
        i. **Return** i.
3. **Return** -1.

Note that this algorithm does more than simply say that a matching value was found or not found in the given list. It returns the index at which the value was found in the list or returns the value -1 to indicate that a matching value was not found.

The following code implements this algorithm in Java.

```
public static int linearSearch(int[] list, int value) {
	for (int i = 0; i < list.length; i++) {
		if (value == list[i]) {
			return i;
		}
	}
	return -1;
}
```

This linear search method can be invoked as shown in this example code segment, which searches the given list of integers for the value 100:

Code	Output

`int[] list = { 7, 1, 9, 5, 11 };`
`if (linearSearch(list, 100) > -1) {`
`      System.out.println("Item found");`
`} else {`
`      System.out.println("Item not found");`
`}`
	`Item not found`

Note that the algorithm and implementing code assume that the list to be searched is not null. Passing a null list, as in **`linearSearch(null, 100)`** results in a null-pointer exception. Given that this search method cannot control how it is called, it would be wise to modify the search method as follows:

```
public static int linearSearch(int[] list, int value) {
      if (list == null) {
            return -1;
      }
      for (int i = 0; i < list.length; i++) {
            if (value == list[i]) {
                  return i;
            }
      }
      return -1;
}
```

This version of the method checks the validity of the list before starting the search and, if the list is null, indicates that the value is not found by returning -1. This is more robust because it anticipates potentially bad input and responds appropriately.

Binary Search

If a list has been sorted, binary search can be used to search for an item more efficiently than linear search. Linear search can require up to n comparisons to locate a particular item, but binary search will require at most $\log_2 n$ comparisons. For example, for a list of 1024 (= 2^{10}) items, binary search will locate an item using at most 10 comparisons, whereas linear search may require 1024 comparisons.

In the binary search method, we first examine the middle element in the list, and if this is the desired element, the search is successful. Otherwise we determine whether the item being sought is in the first half or in the second half of the list and then repeat this process, using the middle element of that list.

To illustrate, suppose the list to be searched is as shown here in the left-most column:

1279		
1331		
1373		
1555		
1824		
1898		
1995	1995	**1995**
2002	2002	2002
2335	**2335**	
2665	2665	
3103	3103	

If we are looking for 1995, we would first examine the middle number 1898 in the sixth position. Because 1995 is greater than 1898, we can disregard the first half of the list and concentrate on the second half (see column two). The middle number in this sub-list is 2335, and the desired item 1995 is less than 2335, so we discard the second half of this sub-list and concentrate on the first half (see column three). Because there is no middle number in this sub-list, we examine the number immediately preceding the middle position —the number 1995 — and locate our number. Note that this approach only works if the list is sorted.

The following algorithm specifies this binary search approach for a list of *n* elements stored in an array, *list*[0], *list*[1], . . ., *list*[*n* – 1] that has been ordered so the elements are in ascending order. If *value* is found, its location in the array is returned; otherwise the value *n* is returned.

Binary Search Algorithm:

1. **Receive** a non-null, sorted *list* of values and a target *value*.
2. **If** *list* is null
    a. **Return** -1.
3. **Set** first = 0 and last = length of the list - 1.
4. **Loop** while first <= last
    a. **Set** middle to the integer quotient (first + last) / 2.
    b. **If** *value* < *list*[middle] **then**
        i. **Set** last = middle – 1;
    c. **else if** *value* > *list*[middle] **then**
        i. **Set** first = middle + 1;
    d. **else**
        i. **Return** middle;
5. **Return** -1.

Note that this algorithm adds the safety check for a null list in step 3. The following code implements this algorithm in Java:

```
public static int binarySearch(int[] list, int value) {
      if (list == null) {
            return -1;
      }
      int first = 0;
      int last = list.length - 1;
      while (first <= last) {
            int middle = (first + last) / 2;
            if (value < list[middle]) {
                  last = middle - 1;
            } else if (value > list[middle]) {
                  first = middle + 1;
            } else {
                  return middle;
            }
      }
      return -1;
}
```

Given the same arguments, binarySearch() returns the same answers as linearSearch(), but it does it more efficiently. This may not be noticeable for small lists, but as the lists increase in size, the efficiency will become more and more noticeable.

8.4. Files

In the previous section, we hard-coded the data as array initializer values in the program itself. While this approach works, it also creates a number of problems. First, the program will only graph the particular set of raw data that it hard-codes, which makes it impossible to reuse the program for other data sets. Second, changing the data values for later use would require repeated rewriting and recompiling of the program, which is unacceptably tedious.

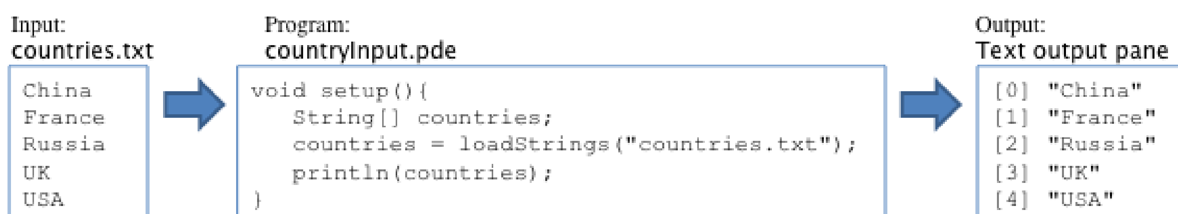
A better approach to this data management task is to separate the data from the program, storing the data in a data file and the program in a program file, and designing the program to read its data from the data file. With this approach, a single program can be used on multiple data files, and the data values can be changed and saved over time.

Files are classified by the kind of data stored in them. Files that contain textual characters (such as the source code for programs or numbers entered with a text editor) are called *text files*. Files that contain non-textual characters (such as the binary code for a compiled program or control codes for a word processor file) are called *binary files*.

This section discusses Processing's capabilities for reading and writing text files.

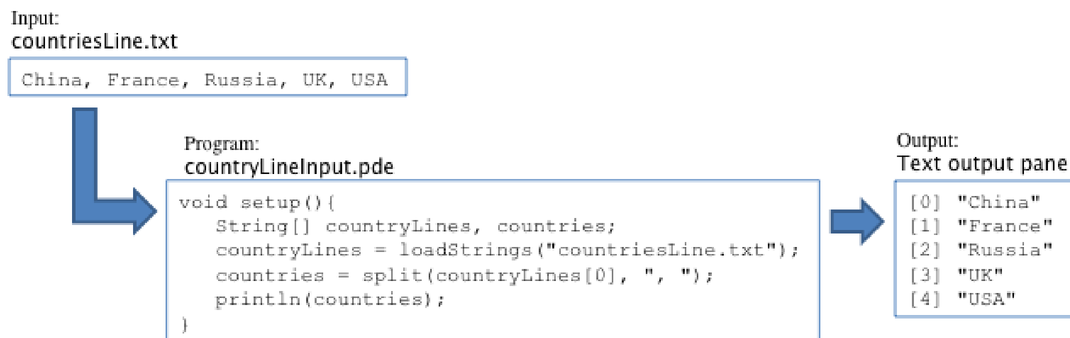
8.4.1. Reading from Files

In Processing, a program reads data from a text file using the `loadStrings()` method. Consider the task of reading a list of country names from a file. The following diagram shows the data file (countries.txt) on the left, the program in the middle and the output on the right.



The data file, `countries.txt`, is a simple text file that contains one country name per line. It is generally stored in the `data` sub-directory. The program uses the `loadStrings()` method to load the lines of the input file into an array of strings. `loadStrings()` automatically reads through the input file and creates an array of strings with one array element per line in the file. The length of the resulting array equals the number of lines in the file. The program then prints the `countries` array to the text output screen. You can see that when Processing prints an array, it automatically includes the array indexes.

In cases where the input file includes more than one atomic value on a given line, the program must split up the input line. In the following example, the country names are listed on a single line in the file.



This program uses `loadStrings()` again but because the input file includes all the country names on one line, `loadStrings()` produces an array of strings with only one element at index 0 whose value is the string:

```
"China, France, Russia, UK, USA"
```

To work with the individual country names, the program must split this one string value into separate country names. It does this using the `split()` method, which takes as arguments: (1) the line to be split, `countryLines[0]`; and (2) a string specifying the characters used to separate the country names, `", "`. The separating string is known as a *delimiter*. This method creates an array of five strings, one for each country with the delimiting characters removed. The result is the same array of country names produced in the last example.

Because these input files are text files, the only type of data that Processing can read from them is string data. To read numeric values from a text file, a program must convert the string value it reads from the file, say "72.961", into the corresponding numeric value for use in numeric computations, 72.961. The following example reads the lines from the file and constructs arrays for the country names and the numeric values for those countries.

Input:
countriesData.txt

```
China 72.961
France 80.657
Russia 65.475
UK 79.425
USA 78.242
```

Program:
countryDataInput.pde

```
void setup(){
   String[] countryLines, countryNames;
   float[] countryValues;
   countryLines = loadStrings("countriesData.txt");
   countryNames = new String[countryLines.length];
   countryValues = new float[countryLines.length];
   String[] tokens;
   for (int i = 0; i < countryLines.length; i++) {
     tokens = split(countryLines[i], " ");
     countryNames[i] = tokens[0];
     countryValues[i] = float(tokens[1]);
     println(i + ": " + countryNames[i] +
             ", " + countryValues[i]);
   }
}
```

Output:
Text output pane

```
0: China, 72.961
1: France, 80.657
2: Russia, 65.475
3: UK, 79.425
4: USA, 78.242
```

This program declares three arrays, a string array for the lines in the file (`countryLines`), a second string array for the country names (`countryNames`) and a float array for the expectancy values (`countryValues`). As with the previous examples, it starts by calling `loadStrings()` to read the lines of the file into an array of strings. This results in an array of 5 strings, the first of which has the following value:

```
"China 72.961"
```

In order to work with the name as a string and the expectancy value as a float, the program must now separate the name string from the float value. This process is *parsing* and the individual elements on the lines being parsed are called *tokens*. In this example, the parsing process will produce data for five countries, with two tokes for each country: a name string and a float value. The program uses the `new` operator to create an empty array for the names and an empty array for the values. Both of these arrays have a length set to the number of lines read from the file. This way, we can add or remove countries from the file and the program will automatically handle the changed number of country lines.

The program then loops through the input and divides each line into the country name portion and the numeric value portion. It does this using the `split()` method discussed in the previous example. In this case, `split()` returns an array of two tokens (`tokens`) , the first of which is a name string (e.g., "China") and the second of which is the expectancy value for that country (e.g., "72.961"). The program stores the name directly into the array of country names. It must then convert the string version of the numeric value (e.g., "72.961") into the corresponding floating point value (e.g., 72.961); it does this using the `float()` conversion method and then loads that converted value into the array of expectancy values.

The program finishes by printing the data on the text output window. This output data looks very much like the input file, but the big difference is that the program has parsed tokens on each line in the file into

values of the appropriate type. This allows the program work with them separately, say to compute the average of the float values or to alphabetize the country names.

8.4.2. Example Revisited

The chapter example can be improved by storing its raw data in a text file. The user could create this text file using a simple text editor or some other tool and then run the program to produce the bar graph output. Here is the data file for this iteration, which includes the year and citation for the data as the first 2 lines.

Data file: `expectancy.txt`

```
2007
GapMinder.com, 2009
China 72.961
France 80.657
Russia 65.475
UK 79.425
USA 78.242
```

The process for reading the data from this file is discussed in this section so we will not include an algorithm here. The following code, when combined with the chart printing code from the last iteration, produces the same output table.

Program file: `ExpectancyTable.pde`

```
/**
 * ExpectancyTableFile displays a textual table of part of
 * GapMinder's life-expectancy data for 2007 (see http://www.gapminder.org/).
 * This version reads its data from a file.
 *
 * @author kvlinden, snelesen
 * @version Fall, 2011
 */
final String filename = "expectancy.txt";

void setup(){
   String[] expectancyLines, expectancyCountries;
   float[] expectancyValues;
   String year, source;

   // Load the data from a file
   expectancyLines = loadStrings(filename);
   expectancyCountries = new String[expectancyLines.length - 2];
   expectancyValues = new float[expectancyLines.length - 2];
   String[] tokens;
   year = expectancyLines[0];
   source = expectancyLines[1];
   for (int i = 2; i < expectancyLines.length; i++) {
      tokens = split(expectancyLines[i], " ");
      expectancyCountries[i - 2] = tokens[0];
      expectancyValues[i - 2] = float(tokens[1]);
   }

   // Print the table as described in the previous iteration...
}
```

This program produces the same output as shown in the previous iteration. The key difference is that it reads its raw data, including the year and data source, from a text file rather than hard-coding it in the program itself. The file input code uses the techniques described in the previous section to load country names and expectancy values from the `expectancy.txt` file. This method treats the first two lines of the text file as the year and the data source, and includes these values in the graphical output.

8.4.3. Writing to Files

In Processing, a program writes data to a text file using the `saveStrings()` method. This method can be viewed as the inverse of the `loadStrings()` method in that it writes the values in an array of strings out to a file rather than reading them in from the file. The following program writes a hard-coded list of country names to a file.

Program:
countryOutput.pde

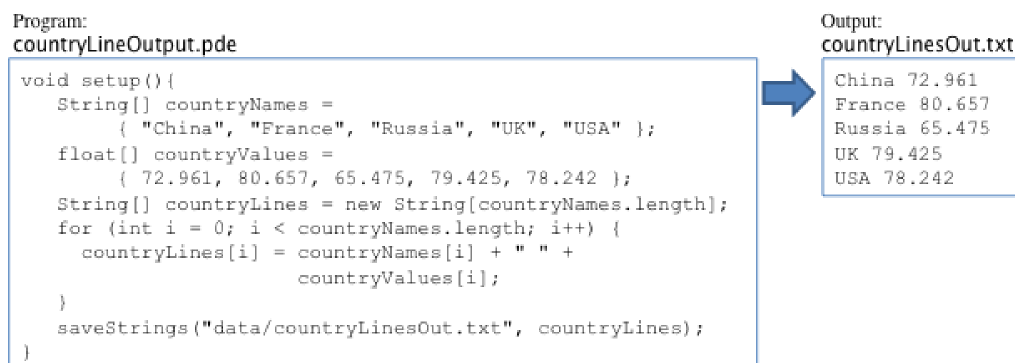
```
void setup(){
   String[] countries =
       { "China", "France", "Russia", "UK", "USA" };
   saveStrings("data/countriesOut.txt", countries);
}
```

Output:
countriesOut.txt

```
China
France
Russia
UK
USA
```

This program saves the countries array in the specified file, one element per line. Note that while Processing automatically looks for input files either in the `data` sub-directory or the program directory itself, it doesn't automatically save output files there so the program must add the file path relative to the program directory, e.g., `data/`, to the output filename. Note also that if a file with the given name already exists, Processing will overwrite it.

`saveStrings()` simply saves the elements of an array of strings, one per line, in a file. If a program wants to combine more than one data value per line, or to include numeric data in its output, then it must create an array of strings matching the format it desires. The following program starts with our country name and expectancy value arrays and produces an output file with one line per country.


Program:
countryLineOutput.pde

```
void setup(){
   String[] countryNames =
        { "China", "France", "Russia", "UK", "USA" };
   float[] countryValues =
        { 72.961, 80.657, 65.475, 79.425, 78.242 };
   String[] countryLines = new String[countryNames.length];
   for (int i = 0; i < countryNames.length; i++) {
     countryLines[i] = countryNames[i] + " " +
                       countryValues[i];
   }
   saveStrings("data/countryLinesOut.txt", countryLines);
}
```

Output:
countryLinesOut.txt

```
China 72.961
France 80.657
Russia 65.475
UK 79.425
USA 78.242
```

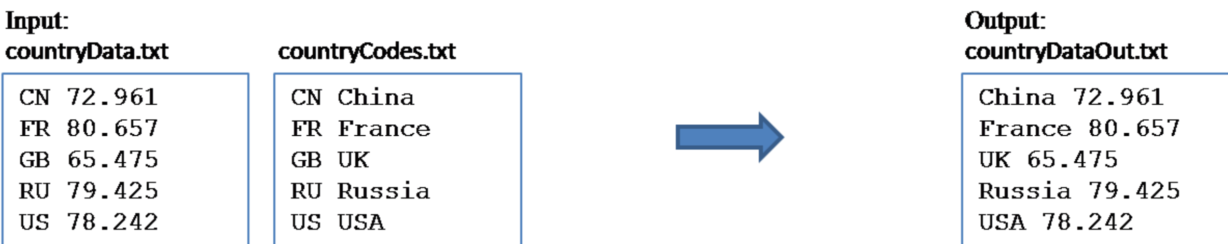
This program combines the raw data stored in its `countryNames` and `countryValues` arrays, and saves the resulting lines in `countryLinesOut.txt`. It uses the string concatenation operator (+) to create a string combining the name and a string version of the numeric value. Note that while in parsing, our program needed to explicitly convert a string value into a float value, Processing's concatenation

operator automatically converts floats into strings. We've used this feature when printing numeric values to the text output panel, and it is useful again here for writing to a text file.

8.4.4. Example Revisited

In the previous iterations, the chapter example code assumes that the country names in the data files are consistent. In practical information systems this is rarely the case. Data comes from a variety of sources (e.g., databases, spreadsheets, web sites) with myriad formats and information systems programmers spend considerable effort systematizing data formats and cleaning up incorrect or missing data. This task, commonly known as *data cleansing*, is critical to the correct operation of data processing programs.

For example, consider the name of the country known above as "China". Will the data represent it as "China", "The People's Republic of China", "China, People's Republic of", "PRC", or "Zhong Guo"? This naming problem is significant enough that the International Standards Organization (ISO) has systematized a two-character coding scheme for country names. China is unambiguously coded as "CN", France as "FR" and so forth. This standard allows our chapter prototype to store its data in a more systematized format, but this format is based on a country code that is harder for users to read. To address this problem, this iteration writes a utility program that reads a raw country data file that uses the ISO country code and writes a new data file that is compatible with the chapter example prototype developed in the last section, including the more readable ISO country name, as shown here:



We know how to read the data in these two input files into arrays, but we now need to do a sort of "lookup" task in which we take the country code from `countryData.txt` and replace it with the corresponding country name from `countryCodes.txt`. This is called *search*. The following algorithm specifies a search method that looks through the country codes loaded from `countryCodes.txt` to find the name of a country with a particular code.

Given:

- `countryCodeLines` is set to an array of strings of the form `"CN China"` where the first two characters are the ISO country code, then there is a space followed by the full name of the country.

Algorithm (for `findCountryName()`)**:**

1. Receive from the calling program a string `countryCode` representing the ISO code of a country and an array `countryCodeLines` representing a list of country code-name pairs as specified above.
2. Repeat for a counter `i` ranging from 0 to the number of country codes:
    a. If `countryCode` = the current country code:
        i. Return the current country name.
3. Return `countryCode`.

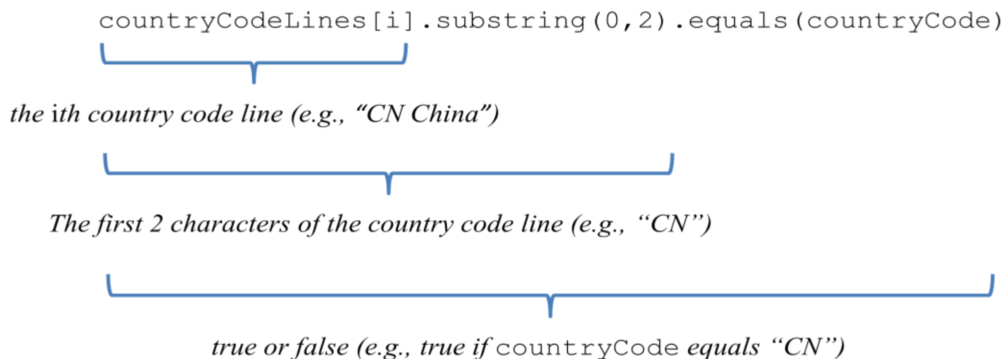
This search algorithm looks for the given country code in the given list of code-name pairs. As soon as it finds a match, it terminates its search loop by returning the country name. If it loops all the way through the code-name pairs without finding a match, it returns the original country code as a last resort. The following program includes an implementation of this algorithm.

```
/**
 * CountryNameConversion replaces the ISO country code in
 * countryData.txt with the official country name specified
 * for that code in countryCodes.txt.
 *
 * @author kvlinden, snelesen
 * @version Fall, 2011
 */

void setup() {
  String[] countryISOCodes, countryData, countryOutputLines;
  countryISOCodes = loadStrings("countryCodes.txt");
  countryData = loadStrings("countryData.txt");
  countryOutputLines = new String[countryData.length];
  String[] tokens;
  for (int i = 0; i < countryData.length; i++) {
    tokens = split(countryData[i], " ");
    countryOutputLines[i] =
        findCountryName(tokens[0], countryISOCodes) + " " + tokens[1];
  }
  saveStrings("data/countryDataOut.txt", countryOutputLines);
}

/**
 * Finds the country name associated with a given country code
 *
 * @param countryCode - the country code to search for
 * @param countryCodeLines - the table of code-name pairs
 * @return the country name or the original country code
 *         if no name is found.
 */
String findCountryName(String countryCode, String[] countryCodeLines)
{
  for (int i = 0; i < countryCodeLines.length; i++) {
    if (countryCodeLines[i].substring(0,2).equals(countryCode)) {
      return countryCodeLines[i].substring(3,
                                 countryCodeLines[i].length());
    }
  }
  return countryCode;
}
```

This program reads two files: `countryData.txt`, which includes a standardized data format that might be found in a common data repository, and `countryCodes.txt`, which includes a mapping

from the ISO country codes to more readable country names for the bar graph output. The program then uses the techniques discussed in the previous section to create a new output string array and fills it with the same data loaded from `countryData.txt` except that it replaces the country code with the more readable name found in `countryCodes.txt`. The resulting file is compatible with the bar graph program built in the last iteration.

This program introduces methods to the simpler code used in the previous iterations. It does this so that it can encapsulate the `findCountryName()` method discussed above. This method uses string manipulation methods to break up the country code strings. As discussed above, the county code strings are of the form `"CN China"` so the following Boolean expression determines whether the given `countryCode` is equal to the `i`th country code string.



This is not an efficient way to search large lists, but it works well enough for this chapter.

8.5. Multi-Dimensional Arrays

The previous sections work with arrays that represent sequences of homogenously-typed values. Each array can be viewed as storing a single, one-dimensional "row" of data. Processing also allows programmers to create multi-dimensional arrays. For example, a two-dimensional array would have "rows" and "columns", a three-dimensional array would have "rows", "columns" and "ranks", and so forth.

This section describes how to declare, initialize and work with multi-dimensional arrays, focusing in particular on two-dimensional arrays.

8.5.1. Declaring and Initializing Multi-Dimensional Arrays

Processing implements a two-dimensional array as an "array of arrays". The following program declares and initializes a two-dimensional array of integers:

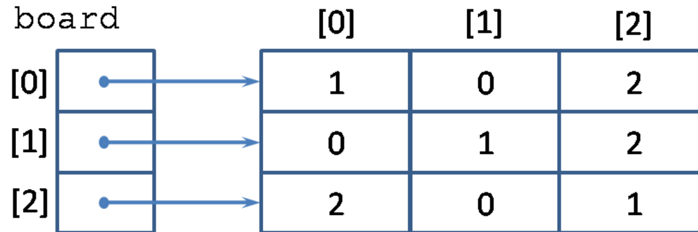
```
int[][] myTable = new int[3][3];
```

This code specifies two sets of square braces for both the type (`int[][]`) and the constructor (`new int[3][3]`), which tells processing to create an array of arrays of integers. The following code

constructs the same data structure but initializes it with data representing the current state of a game of Tic-tac-toe:

```
int[][] board = { { 1, 0, 2 },
                  { 0, 1, 2 },
                  { 2, 0, 1 } };
```

In this board, `0` represents an open cell, `1` represents player X and `2` represents player O, so we can see that player 1 (X) has won the game. This data structure can be viewed as follows:



The `board` object is an array of length 3 whose elements are all arrays of length 3. The one-dimensional array subscript operator works as it did before: `board[0]` refers to the first element of the board array, which is itself a three-element array with values 1, 0, and 3; `board[0][0]` refers to the first element of `board[0]`, which is an integer with value 1. The pattern for using this operator to access the element of the array *anArray* at row *row* and column *column* is shown here:

```
anArray[row][column]
```

Processing supports higher dimension arrays by adding more square bracket pairs.

8.5.2. Working with Multi-Dimensional Array Elements

In the last section, we used a counting `for` loop to access the elements of one-dimensional arrays. In this section, we will use two nested `for` loops to access the elements of a two-dimensional array. For example, the following method initializes every cell in a board to 0 :

```
void initializeBoard(int[][] board) {
  for (int i = 0; i < board.length; i++) {
    for (int j = 0; j < board[i].length; j++) {
      board[i][j] = 0;
    }
  }
}
```

This method uses two nested for loops to loop through the rows (index `i`) and columns (index `j`). The outer loop visits rows 0 through `board.length`, which is the length of the board array (3 for Tic-tac-toe boards). The inner loop visits columns 0 through `board[i].length`, which is the length of the current row at that point in the outer loop.

The following code draws a Tic-tac-toe board consistent with the player tokens specified in the board array parameter.

Code	Output

<pre>public static void printBoard(int[][] board) {
      for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                  if (board[i][j] == 1) {
                        System.out.print("X ");
                  } else if (board[i][j] == 2) {
                        System.out.print("O ");
                  } else {
                        System.out.print(" ");
                  }
            }
            System.out.println();
      }
}</pre>
	<pre>X O
	  X O
	O X</pre>

This code uses two nested `for` loops again to visit each board cell, and then uses an `if` statement to determine which player token, if any, to write in a given cell.

Higher dimensional arrays are processed in an analogous manner, adding one nested `for` loop for each new dimension.

Processing's `loadStrings()` and `saveStrings()` methods are designed for loading and saving one-dimensional string arrays. Handling higher-dimensional arrays requires special programming to split file lines apart into separate data values; this was demonstrated in the previous section.

8.5.3. Example Revisited

The previous iterations on the chapter example have displayed life expectancy data for a single year (2007). We can represent the data over time as well, which requires the following two-dimensional array:

```
float[][] expectancyValues =
   //  China    France   Russia   UK       USA
   { { 70.426, 78.640, 66.790, 77.218, 76.810 }, // 1997
     { 72.028, 79.590, 65.010, 78.471, 77.310 }, // 2002
     { 72.961, 80.657, 65.475, 79.425, 78.242 }  // 2007
    };
```

In this two-dimensional array, each row represents the life expectancy values for each of the five countries in a given year and each column represents the values for a given country over time. For example, `expectancyValues[1][2]` represents the life expectancy for infants born in Russia in 1997 (65.010).

The following code segment reads the names of the countries, the years and the raw data values.

Input:

expectancyValues.txt

```
70.426 78.640 66.790 77.218 76.810
72.028 79.590 65.010 78.471 77.310
72.961 80.657 65.475 79.425 78.242
```

expectancyCountries.txt

```
China
France
UK
Russia
USA
```

expectancyYears.txt

```
1997
2002
2007
```

Program:

```
// Load the expectancy data.
expectancyCountries = loadStrings("expectancyCountries.txt");
expectancyYears = loadStrings("expectancyYears.txt");
expectancyLines = loadStrings("expectancyValues.txt");
expectancyValues = new float[expectancyLines.length][expectancyCountries.length];
String[] tokens;
for (int i = 0; i < expectancyLines.length; i++) {
  tokens = split(expectancyLines[i], " ");
  for (int j = 0; j < tokens.length; j++) {
    expectancyValues[i][j] = float(tokens[j]);
  }
}
// Print a chart of the expectancy data.
println("Average Life Expectancy by Year and Country");
print("\t");
for (int i = 0; i < expectancyCountries.length; i++) {
  print(expectancyCountries[i] + "\t");
}
println();
for (int i = 0; i < expectancyValues.length; i++) {
  print(expectancyYears[i] + "\t");
  for (int j = 0; j < expectancyValues[0].length; j++) {
    print(expectancyValues[i][j] + "\t");
  }
  println();
}
```

Output:

Output text pane

```
Average Life Expectancy by Year and Country
          China     France    Russia    UK        USA
1997      70.426    78.640    66.790    77.218,   76.810
2002      72.028    79.590    65.010    78.471    77.310
2007      72.961    80.657    65.475    79.425    78.242
```

Here, the program assumes that the data is formatted properly. The data values are stored in two dimensional format in the text file (`expectancyValues.txt`) with one row year and one column per country. The number of rows much match the number of years (`expectancyYears.txt`) and the number of columns must match the number of countries (`expectancyCountries.txt`). You can see in the data that four of the five the countries increased their average life expectancy in each year represented in the data (Russia decreased from 1997 to 2002).

8.6. Revisiting the Example

As a final iteration of the chapter example, we introduce graphical bar graphing and animate the bar graph to show the changes of the data over time. Each frame in the animation shows the full bar chart at a particular year and the animation moves from the years furthest in the past to more recent years.

Input:

expectancyValues.txt

```
42.115 50.654 44.100 38.977 42.858 48.051
43.515 53.559 44.600 40.973 45.083 51.016
44.510 56.155 45.000 41.974 47.800 50.350
44.916 58.766 46.218 42.955 50.338 49.849
46.684 59.339 44.020 44.501 51.744 51.509
48.091 59.285 23.599 39.658 53.556 48.825
49.402 54.407 36.087 43.795 55.373 44.578
50.725 50.992 43.413 45.936 56.369 47.813
52.947 54.110 46.242 48.159 58.556 51.542
```

expectancyCountries.txt

```
Ethiopia
Kenya
Rwanda
Somalia
Sudan
Uganda
```

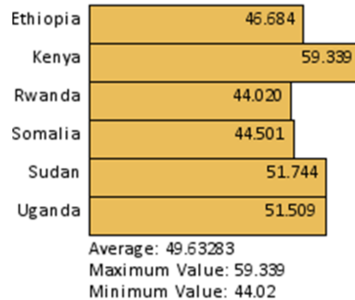
expectancyYears.txt

```
1967
1972
1977
1982
1987
1992
1997
2002
2007
```

Output Frames:

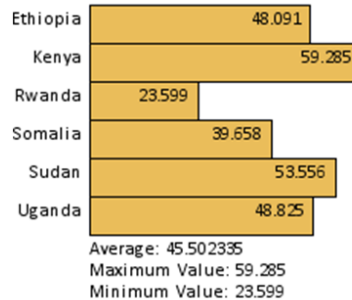
Frame 5:

Average Life Expectancy - 1987



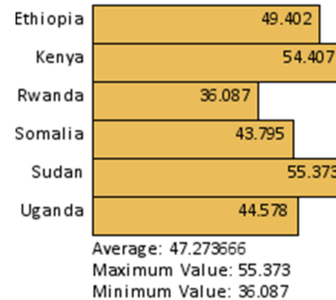
Frame 6:

Average Life Expectancy - 1992



Frame 7:

Average Life Expectancy - 1997




```
final String SOURCE = "GapMinder.com, 2009";
final int BAR_HEIGHT = 25, BAR_WIDTH_UNIT = 3, LABEL_WIDTH = 50, MAX_AGE =
90, HEADER_SIZE = 25, FOOTER_SIZE = 100;

int frameCount;
PFont headerFont, font;
String[] expectancyLines, expectancyCountries, expectancyYears;
float[][] expectancyValues;

void setup() {
  loadExpectancyData("expectancyValues.txt",
                     "expectancyCountries.txt",
                     "expectancyYears.txt");
  int width = BAR_WIDTH_UNIT * MAX_AGE + LABEL_WIDTH + 1;
  int height = BAR_HEIGHT * expectancyCountries.length + FOOTER_SIZE;
  size(width, height);
  headerFont = loadFont("Calibri-Bold-14.vlw");
  font = loadFont("Calibri-12.vlw");
  frameRate(1);
  frameCount = 0;
}
```

```
void draw() {
  background(255);
  drawHeader("Average Life Expectancy in Years (" +
             expectancyYears[frameCount] + ")");
  drawBarGraph(expectancyValues[frameCount], expectancyCountries);
  drawSummaryStatistics(expectancyValues[frameCount]);
  frameCount++;
  if (frameCount >= expectancyYears.length) {
    noLoop();
  }
}
void drawHeader(String headerText) {
  textFont(headerFont);
  fill(0);
  textAlign(LEFT, TOP);
  text(headerText, 10, 5);
}
void drawBarGraph(float[] values, String[] labels) {
  textFont(font);
  textAlign(RIGHT, TOP);
  for (int i = 0; i < values.length; i++) {
    // Draw the bar
    fill(234, 189, 90);
    rect(LABEL_WIDTH, i * BAR_HEIGHT + HEADER_SIZE,
         values[i] * BAR_WIDTH_UNIT, BAR_HEIGHT);
    // Write the label and numeric value.
    fill(0);
    text(labels[i], LABEL_WIDTH - 5, i * BAR_HEIGHT + 5 + HEADER_SIZE);
    text(values[i], LABEL_WIDTH + values[i] * BAR_WIDTH_UNIT - 5,
         i * BAR_HEIGHT + 5 + HEADER_SIZE);
  }
}
void drawSummaryStatistics(float[] values) {
  textFont(font);
  fill(0);
  textAlign(LEFT, TOP);
  text("Average: " + computeAverage(values),
       LABEL_WIDTH, height - FOOTER_SIZE + HEADER_SIZE + 12);
  text("Maximum Value: " + computeMaximum(values),
       LABEL_WIDTH, height - FOOTER_SIZE + HEADER_SIZE + 24);
  text("Minimum Value: " + computeMinimum(values),
       LABEL_WIDTH, height - FOOTER_SIZE + HEADER_SIZE + 36);
  text("Data Source: " + SOURCE, LABEL_WIDTH,
       height - FOOTER_SIZE + HEADER_SIZE + 55);
}
float computeAverage(float[] values) {
  if ((values == null) || (values.length <= 0)) {
    return 0.0;
  }
  float sum = 0.0;
  for (int i = 0; i < values.length; i++) {
    sum += values[i];
  }
  return sum / values.length;
}
```

```
float computeMaximum(float[] values) {
  if ((values == null) || (values.length <= 0)) {
    return 0.0;
  }
  float maximum = Integer.MIN_VALUE;
  for (int i = 0; i < values.length; i++) {
    if (values[i] > maximum) {
      maximum = values[i];
    }
  }
  return maximum;
}

float computeMinimum(float[] values) {
  if ((values == null) || (values.length <= 0)) {
    return 0.0;
  }
  float minimum = Integer.MAX_VALUE;
  for (int i = 0; i < values.length; i++) {
    if (values[i] < minimum) {
      minimum = values[i];
    }
  }
  return minimum;
}

void loadExpectancyData(String valuesFilename, String countriesFilename,
String yearsFilename) {
  expectancyCountries = loadStrings(countriesFilename);
  expectancyYears = loadStrings(yearsFilename);
  expectancyLines = loadStrings(valuesFilename);
  expectancyValues = new
float[expectancyLines.length][expectancyCountries.length];
  String[] tokens;
  for (int i = 0; i < expectancyLines.length; i++) {
    tokens = split(expectancyLines[i], " ");
    for (int j = 0; j < tokens.length; j++) {
      expectancyValues[i][j] = float(tokens[j]);
    }
  }
}
```

This code is largely copied from the previous iterations, with two significant additions:

- The `drawBarGraph()` method formats the raw data in the form of a bar graph.
- The `draw()` method runs repeatedly based on the `frameCount` variable.

The three particular frames shown above show the effect of the Rwandan civil war (1990-1993) on the average life expectancy in the early 1990s.