# Section 1 - Declaring Arrays

### Introduction to Week Eight

We have skipped one of the most useful tools in programming, arrays, because we wanted to move fast and get into some interesting Java. Now that we have so many tools at our disposal, we can add arrays and really have some fun with them.

With arrays we can quickly declare large numbers of data objects organized in a single group. They can be used more efficiently than if we had declared hundreds of separate, unrelated objects.

### Reading

As usual, first study this module completely. Then, if you want more, read the (single dimensional) array material in the text.

## 8A.1.1 What is an Array?

An ***array*** is a group of objects that are positioned in contiguous memory locations that have the same ***root name***, and are distinguished from each other by an ***index***. If the ***root name*** of the array was **student** and we wanted **20** variable objects of type **int** in this array, we would declare the array as follows:

```
int[] student;
```

and we would allocate the variables comprising the array with this notation:

```
student = new int[20];
```

meaning that we desire **20 int** objects, stored in consecutive memory locations, and the names of the individual elements are to be:

```
student[0]
student[1]
student[2]
.
.
.
student[19]
```

Note that the numbering begins with 0, and ends with 19. The elements are stored one after another, starting with **student[0]** as the first, or smallest, location in memory.

The variable **student[0]** is called an ***element*** of the array, but remember that its value can be controlled individually as if it were a separate object declared individually:

```
student[0] = 25;
System.out.println( student[0] );
```

The integer inside the brackets is the ***array index***. It can be a constant or integer expression. Be careful not to confuse the ***size of the array***, which is a constant inside the brackets in the ***allocation*** statement:

```
 student[] = new int[20];
```

with the ***index of the array***, which must be in the range from **0** to **[size of array] - 1**, or **19** in this case.

Usually, each ***array element*** is modified in a way that brings out the group nature of arrays. For example, to set every array element to 7, we could use the following loop:

```
for (k = 0; k < 20; k++)
   student[k] = 7;
```

By using a **k** as an index, you can run through all of the array ***elements*** in a single ***for*** statement. As simplistic as it sounds, this for loop demonstrates the main characteristics of how arrays are used in programs. It shows how natural it is to marry arrays with for loops.

# 8A.1.2 The Object Nature of Arrays

Although the above array might consist of individual primitive data elements (**ints**), the array itself behaves like an object. Observe how the array is declared and allocated:

```
int[] student;
student = new int[20];
```

This reveals that an array name acts exactly like a class reference, and needs to be pointing to an allocated object before it can be used. In this case the "object" is the contiguous sequence of elements.

This analogy goes all the way. When we pass an array name to a method, it goes in as a reference, just like an object does. That implies any changes to the array elements inside the method carry back to the **main()**. Similarly, if we return an array as a functional return value of a method, we are returning a reference only.

We will see examples of all of this very shortly.

## 8A.1.3 The Base Data Type of an Array

Above we used **int** as the base data type for the array. However, the base type need not be primitive. It could be **String**:

```
String[] student;
student = new String[50];
```

or some class, user-defined, as in an array of **Employees**:

```
Employee[] worker;
worker = new Employee[100];
```

or Java-defined, as in an array of **JButtons**:

```
JButton[] myButtons;
myButtons = new JButton[13];
```

## 8A.1.4 Initializing an Array

Besides using the for loop we have seen above, we have a short cut way of initializing an array to some programmer-specified values. Here is an example:

```
double[] myArray = {10.2, 56.9, -33, 12, 0, 2,
      4.8, 199.9, 73, -91.2};
```

This list can be as long as you wish, and can go on for several lines. The important thing is that the values match the base type of the array. Here is an example with a different base type:

```
String[] myArray = {"martin", "claudia", "sandra", "samuels",
       "terry", "jack", "clark", "palmer", "abraham", "Mike"};
```

# Section 2 - An Array Example

## 8A.2.1 An Array of Strings

We will create an array of 20 strings representing student names. The declaration follows our rules for arrays, but we combine the lines into one statement:

```
String[] student = new String[20];
```

We then want to place a recognizable initial value into each of the student strings and we do this with a *for loop*:

```
// initialize the array
for (int k = 0; k < 20; k++)
   student[k] = "undefined";
```

***Notice two things about this loop:***

1. It uses the size of the array, 20, to control the upper limit of the loop.
2. It uses a less than, <, relational, not a <= operator.

The second observation is very important. There is no array element **student[20]** in an array of 20 students. The last member of the array is student[19]. Don't ever forget this.

Because it is very bad to use constants throughout the program (like 20, 19 or 100), we can avoid the problem of remembering what the size of the array is by using the special notation **[array name].length**. This is not a method call, **length()** like we have seen with **Strings**, but an actual member of the array class, so to speak. So no parentheses, please. Here is the preferred technique:

```
// initialize the array
for (int k = 0; k < student.length; k++)
   student[k] = "undefined";
```

## 8A.2.2 A New showInputDialog()

To help us with this program, we are going to show an ***overloaded*** version of the **showInputDialog()** that we have not had yet. It takes more parameters, which is a pain, of course, but they are useful. This new version lets us display a string in the title bar of the dialog box. For instance, when we are editing a particular student's name in a **showInputDialog()**, we might like the information of which student we are modifying to appear in the title bar, like so:


This is very compact and understandable to the user. They see the current name of the student in the title bar and now they can modify that name if they wish.

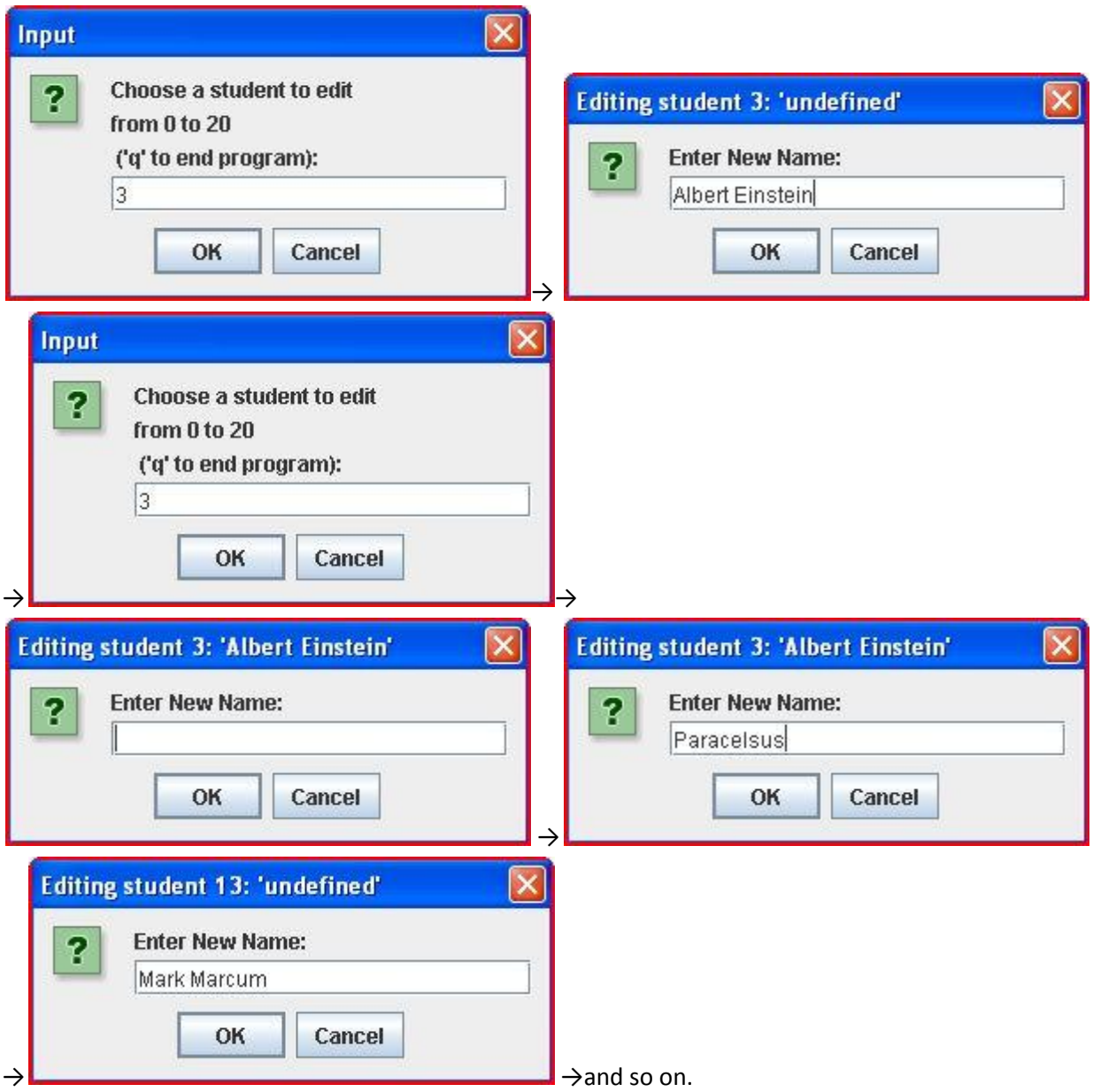
The form of this version of **showInputDialog()** is:

```
newName = JOptionPane.showInputDialog(
   null,
   "Enter New Name:",
   "Editing student " + value + ": '" + student[value] + "'",
   JOptionPane.QUESTION_MESSAGE );
```

There is the familiar null in the first position and the "Enter New Name:" that is displayed in the input box, but in the third position of the argument list we can provide another **String** to appear in the title bar. The fourth parameter in the argument list is a ***static class constant*** of the **JOptionPane** class, and controls the icon that appears in the box. When **QUESTION_MESSAGE** is passed, the icon is a question mark, "?", as shown.

# 8A.2.3 The Example

We will go into a loop that asks the user what student he wants to edit (by selecting a number from 0 to 19), and then, displays an inputDialog box to get a new String name for that student with that number. Here is a sample output:



→and so on.

Notice that the *'q' to quit* prompt is really not handled yet. We'll get there later.

We are ready to try out our first example of arrays. This example is very fragile. It does nothing to protect against bad user input. There are tons of things the user can do wrong here, and if any of them occurs, you will see error messages in your console window and an early termination of the program.

These are called exceptions, and we will learn how to handle them ourselves, later in this lecture. For now, the first, simple version of the program that demonstrates how arrays are used.

```
import javax.swing.*;

public class Sample
{
   public static void main (String[] args)
   {
      String[] student = new String[20];
      int value = 0;
      String strValue;
      String newName;

      // initialize the array
      for (int k = 0; k < student.length; k++)
         student[k] = "undefined";

      // infinite loop until user enters q or cancels
      while (true)
      {
         strValue
            = JOptionPane.showInputDialog("Choose a student to edit\n"
            + "from 0 to " + (student.length-1) + "\n"
            + " ('q' to end program):");

         value = Integer.parseInt(strValue);

         newName = JOptionPane.showInputDialog(null,
            "Enter New Name:",
            "Editing student " + value + ": '" + student[value] + "'",
            JOptionPane.QUESTION_MESSAGE );

         student[value] = newName;
      }
   }
}
```

**Note**

The source code correctly asks for users between 0 and 19, but the screen shots were taken before this fix, and they display the number 20 rather than 19.

# Section 3 - Catching Exceptions

## 8A.3.1 NumberFormatExceptions

We want to present the robust version of the student array program. To make it really air-tight, however, we must add a new type of input filter, the ***Exception***.

Whenever we get an input from the user in the form of a **String** and then convert this to a number with a statement like:

```
   value = Integer.parseInt(strValue);
```

there is always the possibility that this conversion fails due to a bad **strValue**. Can you think of the types of user inputs that would provoke a complaint by the static class method **parseInt()** of the Integer class? Think of at least two different kinds of errors.

If an error occurs during the invocation of this method, Java ***throws a*** **NumberFormatException** for us to ***catch***. This sounds very abstract, but we can make it concrete by seeing how we "keep on the look-out" for this so called ***Exception,*** and how we subsequently ***catch*** it.

```
try
{
   value = Integer.parseInt(strValue);
}
catch (NumberFormatException e)
{
   // typed illegal int ("34.5" or "sdf" or "*2", etc.)
   // take the action you want (showMessageDialog, exit, ...)
}

// if they get here, the conversion was successful
```

It's a lot of code in place of our original one line, but it is needed. In our program, this ***try/catch block*** will be inside an ***input filter loop*** which will continue indefinitely until the user gives us a real integer (i.e., one with no illegal characters). Therefore the proper action to take if the input is bad is to ***continue*** with another pass of the ***filter loop***, asking again for input. If the number passes this test, on the other hand, we will proceed to the next test in the sequence and eventually, if they give us a perfect answer, we will ***break*** from the ***filter loop.***

There are many places where we could have used a ***try/catch block*** but instead ignored the dangers. Whenever we call a method in the Java library we run the risk of it generating an exception because of bad data that we are passing in as parameters. We need to investigate the documentation to see what kind of exception is ***thrown*** in each method and then supply a catch block to handle the exception.

For most numeric conversion methods like **parseInt()** or **parseDouble(),** the ***Exception*** will be this one: **NumberFormatException**.

We don't have to do anything with the parameter **e** that we see in the ***catch phrase*** above. For now just copy the phrase as is. Also, e is a formal parameter and, as such, can be named anything you want, as long as it doesn't conflict with any other variable names in your local method.

# Section 4 - Studentus Robustus

## 8A.4.1 A Complete Student Name Example

Here is the complete example of the student name program that uses arrays, input filter loops and exceptions. Study it so you can use these techniques yourself.

```
import javax.swing.*;

public class Sample
{
   public static void main (String[] args)
   {
      String[] student = new String[20];
      int value = 0;
      String strValue;
      String newName;

      // initialize the array
      for (int k = 0; k < student.length; k++)
         student[k] = "undefined";

      // infinite loop until user enters q or cancels
      while (true)
      {
         // input filter loop to get a valid array index from user
         while (true)
         {
            strValue = JOptionPane.showInputDialog(
                  "Choose a student to edit\n"
                  + "from 0 to " + (student.length - 1) + "\n"
                  + " ('q' to end program):");
            // did they click cancel or close box?
            if (strValue == null)
               return; // if so, end program

            // did they type a string that was not 1 or 2 digits?
            if (strValue.length() < 1 || strValue.length() > 2)
               continue; // if so, prompt again

            // did they type 'q'?
            if (strValue.length() == 1 && strValue.charAt(0) == 'q')
               return;  // if so, exit program
```

```
            // finally, convert to int.  Is it a valid int?
            try
            {
               value = Integer.parseInt(strValue);
            }
            catch (NumberFormatException e)
            {
               // typed illegal int ("34.5" or "sdf" or "*2", etc.)
               continue;
            }

            // is the int within our expected range?
            if (value >= 0 &&  value < student.length)
               break;  // good value, escape from this loop

            // typed a valid int but was it in range?
            // could test inside loop or in while statement
         }

         // infinite loop for name input
         while (true)
         {
            newName = JOptionPane.showInputDialog(null,
               "Enter New Name:",
               "Editing student " + value + ": '"
                     + student[value] + "'",
               JOptionPane.QUESTION_MESSAGE );

            // they hit cancel or close box
            if (newName == null)
               break;
            else if (newName.length() >=2 && newName.length() <= 40)
            {
               student[value] = newName;
               break; // from input loop
            }
            else
               JOptionPane.showMessageDialog(null,
                  "Invalid Name, try again.");
         }
      }
   }
}
```

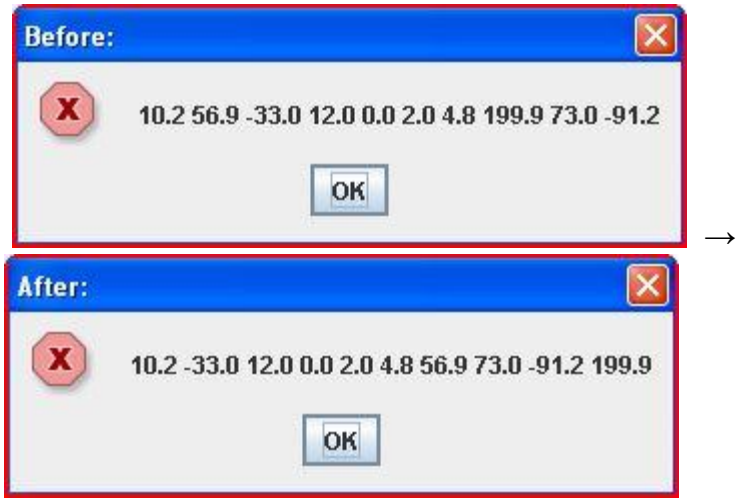
# Section 5- Arrays, Methods and Sorting

## 8A.5.1 Passing Arrays to Methods

Arrays act like ***class objects*** when it comes to ***methods***. If you pass an array into a method as a parameter, any changes made to the array elements will persist after the method returns. This is good if you want the method to modify the array, and bad if you don't.

Let's take advantage of this in a method that will "float" the largest element of a **double** array to the top of the array.

Here is our desired output:




Notice that the 199.9 ended up in the last position on the right, as we desired. Also, the array was modified in other places, as well, but that isn't as important to us.

We have two methods for this program. Since we are not using ***OOP*** right now, we will make them both ***static methods*** and call them from **main()** without any dereference. One method will *float* the largest element of the array to the top. The other method will print out the array for us. Note that we pass the **printArray()** method two arguments: the string to place on the title ("Before: " or "After: ") and the array to print.

Also notice the syntax of the formal parameter list in a method that takes an array:

```
void func( double[] arrayParam )
```

Of course, double could be any ***primitive type*** or ***class name***, and represents the basic type of the array.

**floatLargestToTop()** returns a **boolean true** if a change was made to the array, and **false** if not. We are not going to make use of this return type in our **main()** just yet.

```
public class Foothill
{
   public static void main (String[] args)
   {
      double[] myArray = {10.2, 56.9, -33, 12, 0, 2, 4.8,
            199.9, 73, -91.2};

      printArray("Before: ", myArray);
      floatLargestToTop(myArray);
      printArray("After: ", myArray);
   }

   public static boolean floatLargestToTop(double[] data)
   {
      boolean changed = false;
      double temp;

      for (int k = 0; k < data.length-1; k++)
         if (data[k] > data[k+1])
         {
            temp = data[k];
            data[k] = data[k+1];
            data[k+1] = temp;
            changed = true;
         }
      return changed;
   }

   // print out array with string as a title for the message box
   public static void printArray(String title, double[] data)
   {
      String output = "";

      for (int k = 0; k < data.length; k++)
         output += " "+ data[k];
      JOptionPane.showMessageDialog( null, output, title,
            JOptionPane.OK_OPTION);
   }
}
```

Study the for loop inside the **floatLargestToTop()** method and make sure you understand what's going on. It simply compares adjacent elements in the array and ***swaps*** them if the left one is larger than the right. It does this starting at the bottom of the array and moving to the top, thereby moving the large elements gradually to the right. Besides moving the largest to the top, it also helps move some of the other large elements more to the right (higher), and other small elements, more to the left (lower).

# 8A.5.2 Sorting

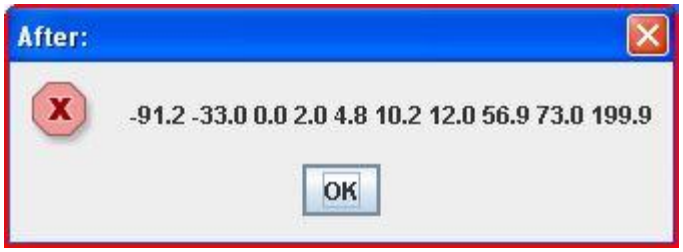
If you followed the relatively simple concept of the method **floatLargestToTop()** then you have actually comprehended a much more sophisticated and universal concept for free: ***sorting***. ***Sorting***, the technique (***algorithm***) for placing arrays in increasing or decreasing order, has long been a topic of computer science classes, and is actually studied in graduate courses. And you are only one statement away from understanding ***sorting***. Here's the statement:

```
for (int k = 0; k < myArray.length; k++)
```

"That's just a for loop control," you say? Right. What I'm telling you is that if you add this one statement to your **main()** method, you will have ***sorted*** the initial array so that it will appear in increasing order. Here's what I mean. We repeat the method call to **floatLargestToTop()** exactly **myArray.length** times, and you will end up with a sorted array.

```
printArray("Before: ", myArray);
for (int k = 0; k < myArray.length; k++)
   floatLargestToTop(myArray);
printArray("After: ", myArray);
```

If you make this change and run the program, you will get this result:




Nice, eh? What happened? Each time we called the method, a new number got floated as far to the top as it could. That is, once the largest number was placed at the end of the list, it did not interfere with the second largest making it almost to the end. Then that number didn't interfere with the third largest making its way to the third-from-top spot, etc.

Before doing anything else, we have to modularize this so that it is contained in a method of its own, **arraySort():**

```
public static void arraySort(double[] array)
{
   for (int k = 0; k < array.length; k++)
      floatLargestToTop(array);
}
```

This important step removes from **main()** the responsibility of having the ***sorting algorithm*** partly in it, and partly in the **floatLargestToTop()** method. We always want to encapsulate a reusable task in a method of its own. Once we do this, **main()** looks, once again, simple:

```
printArray("Before: ", myArray);
arraySort(myArray);
printArray("After: ", myArray);
```

And there you have a fully functional **arraySort()** method that is extremely short.

# 8A.5.3 Improving the Sort Algorithm

We can make this algorithm smarter. For one, what do we know if we called **floatLargestToTop()** and no modifications were made to the array? We would know that the array is done being sorted. This often happens after only a few calls to **floatLargestToTop()** since each call improves the situation quite a bit. So our first improvement is to use the **boolean *return value*** that we have been ignoring up to now:

```
public static void arraySort(double[] array)
{
   for (int k = 0; k < array.length; k++)
      if ( !floatLargestToTop(array) )
         return;
}
```

The next improvement involves the realization that each time we call **floatLargestToTop(),** we can stop a little closer to the beginning of the array. That is, the loop inside **floatLargestToTop()** can get shorter every time we call it. For this, we need a second parameter to that method that tells it how far to go.

Here are the two methods that, together, make up the array sort with this change:

```
// returns true if a modification was made to the array
public static boolean floatLargestToTop(double[] data, int top)
{
   boolean changed = false;
   double temp;

   for (int k =0; k < top; k++)
      if (data[k] > data[k+1])
      {
         temp = data[k];
         data[k] = data[k+1];
         data[k+1] = temp;
         changed = true;
      }
   return changed;
}

 public static void arraySort(double[] array)
 {
    int everShrinkingTop;

    for (    everShrinkingTop = array.length - 1;
             everShrinkingTop > 0;
             everShrinkingTop--
        )
       if ( !floatLargestToTop(array, everShrinkingTop) )
          return;
 }
```

# Section 6 - Sorting String Arrays

## 8A.6.1 Nothing Really New

I know. You're thinking, "another section to read in this module. Can't we have a light week for a change?"

Believe me, this section is one you can read and understand in less than five minutes. Maybe three.

All I want to do here is revise the sorting application we just learned so that it can sort an array of **Strings** instead of **ints**. To do that we have to know how to ask the question ***"is string1 < string2?"***, that is, does **string1** come before **string2** in alphabetical ordering. For that we use the String instance method **compareToIgnoreCase()**.

```
string1.compareToIgnoreCase( string2 )
```

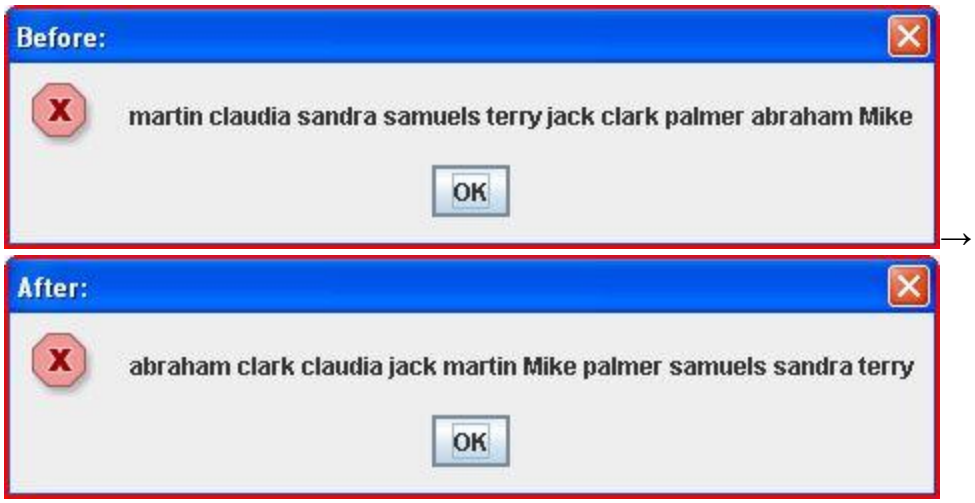
Returns a 0 if they are equal, a negative number if **string1** comes before **string2** and a positive number if **string1** comes after **string2**. So instead of asking whether

```
data[k] > data[k+1]
```

we ask the question:

```
data[k].compareToIgnoreCase(data[k+1]) > 0
```

And with that one change, we can change the program so it works for strings. The output would look like this:




Here is the program with the small modifications to work with Strings.

```
import javax.swing.*;

public class Foothill
{
   public static void main (String[] args)
   {
      String[] myArray = {"martin", "claudia", "sandra", "samuels",
            "terry", "jack", "clark", "palmer", "abraham", "Mike"};

      printArray("Before: ", myArray);
      arraySort(myArray);
      printArray("After: ", myArray);
   }

   // returns true if a modification was made to the array
   public static boolean floatLargestToTop(String[] data, int top)
   {
      boolean changed = false;
      String temp;

      for (int k = 0; k < top; k++)
         if (data[k].compareToIgnoreCase(data[k+1]) > 0)
         {
            temp = data[k];
            data[k] = data[k+1];
            data[k+1] = temp;
            changed = true;
         }
      return changed;
   }

    public static void arraySort(String[] array)
    {
       int everShrinkingTop;

       for (    everShrinkingTop = array.length - 1;
                everShrinkingTop > 0;
                everShrinkingTop--
           )
          if ( !floatLargestToTop(array, everShrinkingTop) )
             return;
    }

   // print out array with string as a title for the message box
   public static void printArray(String title, String[] data)
   {
      String output = "";

      for (int k = 0; k < data.length; k++)
         output += " "+ data[k];
      JOptionPane.showMessageDialog( null, output, title,
            JOptionPane.OK_OPTION);
   }
}
```

See? I told you: three minutes.