10 Arrays and Strings

Read through this section carefully, it contains a significant number of new concepts and difficult syntax. You will almost certainly have to refer back to this section many times while undertaking the final checkpoint(s).

It is also essential that you understand loops before attempting this section, it is strongly suggested that you re-read the section on loops before progressing.

10.1 Introduction

Up to now we have been declaring and using *single* variables mostly of type `int` or `double`. This has restricted you to problems that involve little data. This limitation is addressed by the use of *arrays* which are fundamental to useful scientific programming.

New programmers find *arrays* initially a difficult cencepts but since they are fundamental to all numerical programming, it is essential that you work through this section very carefully and try the given examples.

10.2 One Dimensional Array

The one-dimensional array is an object that contains a set of variables with the same *base name* but with a number of elements, similar to a mathematical *vector*[1].

To declare and allocate an `int` array of length 10 we use,

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

which is actually a two stage process that:

1. Declares an object , `iValues`, that is an `int` array.
2. Allocates storage for 10 `int` in this object, and set their initial value to `0`.

The **two** stage aspect of this process may appear irrelevant at the moment, but will become clear later on.

The 10 `int` variables have *names*

```
iValues[0], iValues[1], iValues[2],...,iValues[8],iValues[9]
```

Note: The index starts at "0", so the last elemenst is "9"!!!

In addition the Object `iValues` has accessible properties, this most important one at this stage being its length, which is accessed by `iValues.length`.

In a program we can either treat the elements of `iValues` as 10 separate integers, or more usefully we can access them in a loop, for example to set all 10 elements to 100 we can use:

```
int iValues[] = new int[10];
for(int i = 0; i < iValues.length ; i++){
    iValues[i] = 100;
   }
```

[1] The numbering of elements is different.

where the indexing variable `i` *must* be an `int`.

Note the use of `iValues.length` *in the termination condition of the* `for` *loop, this is good programming practice as it removes the use of arbitrary numbers in the code.*

Arrays of other data types are identical, for example `double` arrays are declared as:

```
double xValues[] = new double[10];
```

which will declare an object which is a `double` array, allocate space for 10 elements, and set their initial values to `0.0`.

Again the individual elements of the array can be accessed via a `int` variable index exactly like the `int` array discussed above.

Unlike many other computing languages in JAVA you can declare and/or allocate objects at any point in your program and more importantly for arrays, the size can be an `int` variable. So for example you can read in the length of the array from a `Display` object and use this value to allocate an array of the correct length, a typical code fragment would be,

```
int lengthOfArray;
Display myPanel = new Display("Dynamic Arrays");
Input arrayLengthInput = new Input("Length of array : ",10);
myPanel.addInput(arrayLengthInput);

while(true) {
      myPanel.waitForButtonPress();
      lengthOfArray = arrayLengthInput.getInt();

      int iValues[] = new int[lengthOfArray];   // Declare and allocate

      <- Rest of code using int array iValues of length iValue.length ->
      }
```

which reads the length of the array, then declares and allocates it.

The obvious questions is "what happens the second time round the loop" when you ask for a different size array. Again the power of JAVA comes to your aid and the "old array" will automatically be deallocated[2] and the new one created of the new length. This feature of JAVA is wonderful for programmers but is also one of the main reasons why JAVA code is always much slower than other programming languages.

10.3 Initialisation and Constant Arrays

In addition to the allocation scheme detailed above, arrays can be initialised at definition, for example `int` and `double` arrays

```
int iValues[] = {1,9,5,3,4};
double xValues[] = {12.78,23,9.25262,1.2e6,9.45e-3};
```

[2]This is in stark contrast to C and "C++" where the programmer must deal with all the allocation and deallocation themselves and the complications and potential for error this involves!

declared and allocates objects with **5** elements and will have the values of their elements set to the given list.

This is actually of most use when declaring *constant* arrays, where the values *cannot* be changes in the program, in this case the additional keyword `final` is used, for example,

```
final double coefficients[] = {1.454563879 , 0.454536723e-4,
           0.455428945e-9 };
```

will form a constant `double` array of length 3.

10.4 Addressing Arrays

The processing of arrays is almost always done with some type of loop, the most common in scientific computing being the `for` loop we have seen earlier. For example if we want to fill a `x[100]` and `y[100]` arrays with the values of $y = \sin(x)$ for x in the range $0 \rightarrow 5\pi$ we would use:

```
double x[] = new double[100];
double y[] = new double[100];

for(int i = 0; i < x.length; i++) {
    x[i] = 5.0*Math.PI*(double)i/(double)x.length;
    y[i] = Math.sin(x[i]);
   }
```

which goes round the `for` loop $\times 100$, calculates the value for `x[i]` then uses its value to calculate `y[i]`.

This will be the most common loop and array construct in your programs, and in particular in the next *checkpoint*.

Note: the element of an array can only be accessed via an `int`.

10.5 Multi-Dimension Arrays

In many scientific applications we will want to use multi-dimensional arrays, for example a matrix is best represented by a two-dimensional array.

A two dimensional `double` array of size 8×10 elements is defined by

```
double matrix[][] = new double[8][10];
```

which is actually **8** one-dimensional arrays each of length **10**.

This would be accessed by *two* indexing variables, typically in a nested `for` loop, for example to set the whole array `matrix` to 100 we would use:

```
for(int j = 0; j < matrix.length; j++) {
      for(int i = 0; i < matrix[j].length; i++) {
            matrix[j][i] = 100;
          }
    }
```

Note that `matrix.length` will return the **first** dimension, (8 in this case), while `matrix[j].length` gives the **second** dimension (10 in this case).

I have avoided denoting the indices of the array with the misleading terms *row* and *columns* since there is an unfortunate, and irreconcilable difference, between mathematics and computing regarding the meaning of these terms, so when using two-dimensional arrays to perform matrix operations, great care must be used to get the indices the *right way round*; see the example below.

As with one-dimensional arrays the allocation size can be `int` variable set as run-time. It is also "possible" to allocate two-dimensional array which is an array of one-dimensional arrays with *different* lengths. This is a recipe for vast confusion and should be avoided!

Higher dimensional arrays are obvious, for example a three-dimensional `double` array can be declared and set to a constant with:

```
double tensor[][][] = new double[3][5][6];

for( int k = 0 ; k < tensor.length ; k++) {
   for( int j = 0 ; j < tensor[k].length ; j++) {
      for( int i = 0 ; i < tensor[k][j].length ; i++) {
            tensor[k][j][i] = 100;
         }
     }
   }
```

If you have understood the section above, then you will guess that what you have *really* created is **3** two-dimensional arrays, each of which consists of **5** one-dimensional arrays, each of which has **6** elements. If not do not worry about it at this stage, you will come back and visit this problem in next year's course!

You can continue this "dimension expansion" to any order you like, but arrays of dimensional greater than three are unlikely to be very useful!

10.6 Reading Arrays with the `Display` class

Up to now you have been using the `Input` class and methods to read single integers and doubles, with one prompt per input field. This is not convenient when reading array elements, so there are additional methods to support this, these being

```
Input(String prompt , double doubleArray[]);
Input(String prompt , int intArray[]);
```

that take one String prompt followed by an `double` or `int` array containing the default values. When added to the `Display` this forms an input line with one prompt and an input field for each element of the array.

The input fields can then be read by the methods

```
getDouble(int field);
getInt(int field);
```

that reads a `double` or `int` from the input field specified by `field`. Note for a array of length N elements, the input fields and numbered $0 \rightarrow N-1$, or more compactly using the methods

```
int[] getIntArray();
double[] getDoubleAttay();
```

which will return a one-dimensional array of the same length and the default array given when the `Input` object was created.

The following partial example sets up an `Input` to read three `double`s form a single input field. After the `waitForButtonPress()` the three `double`s are then read back in from their respective fields.

```
import display.*;

public class Coordinate{
  public static void main(String args[]){

    Display myDisplay = new Display("Read a coordinate");
    double coordinate[] = new double[3];    // Defaults to 0.0
     /*
     *           Form input for doubles with three input fields
     */
    Input coordInput = new Input("Give coordinate", coordinate);
    myDisplay.addInput(coordInput);

    myDisplay.waitForButtonPress();

    coordinate = myDisplay.getDoubleArray();

    < ... rest of code ...>
```

This scheme is very useful for reading short array, longer arrays conatining many 10s, 100s, or 1000s of elements are typically read from files, which are not covered in this initial course, however since this is so useful there is an additional optional section in this booklet that covers basis file handling. Look at this *after* you finish this course.

10.7 Warning

When you declare an array to have "`100`" elements it is up to **you** to make sure your program, or any functions your program calls, does not try to access elements outwith the range $0 \rightarrow 99$. JAVA has strict array bounds checking, and if you do try and access elements outside the array your program will crash with an "exception" of the type:

```
java.lang.ArrayIndexOutOfBoundsException
```

and a line number where this exception (error!) occurred.

10.8 Strings

`String`s are the main object in JAVA used for handling non-numeric data. In scientific (numeric) applications this usually means for formatted output, filename, input parameters and other relatively simple tasks. At this stage of your programming career you need to know relatively little about the details of Strings, other than how to use them.

We have been using `String`s right from the first program mainly for data output and passing title and other fixed non-numeric data to objects. This is all you actually need at this stage, but the String is a nice example of an object and how to access some of the methods that act on strings will help you with your future programming.

10.9 Declaration and Initialisation

As we have seen we can declare and inititalise a `String` with,

```
String outputString = "Hello World !";
```

which declares `String` called `outputString` and sets its value to `Hello World !`, being 13 characters long. A few points to note:

1. the `""` are **not** part of the `String`, they mark the beginning and the end.
2. Spaces, punctuation, operators etc. inside the `String` are just characters and have no special meaning.
3. To put a `"` inside a String you must use \\`"`, so the declaration,

   ```
   String sentance = "She said \"My name is Jane\"."
   ```

   sets the string value to,

   ```
   She said "My name is Jane".
   ```

   which has 27 characters. Note the \\`"` is a single character.

Constant Strings
Again if we have a `String` in our program that **must** not be altered or added to, this can be declared as `final`. Any attempt to accidentally alter this String will the result in a program crash.

10.10 Extending Strings

As we have seen in previous sections we have simple "add" `String`s together to form a single `String`, so for example we can write.

```
String outPut = "";                  // Declare a null string
String firstWord = "Hello";
String secondWord = "World";

outPut += firstWord + " " + secondWord + " !";
```

A few details points are worth noting here:

1. The declaration of `outPut` sets its value to a blank String that contains no characters. This may appear odd, but it does allow easy use of the += operator to "add" Strings.
2. The expression forming `outPut` consists of the "addition" of **four** Strings.

In addition, as we have seen previously can "add" `ints` and `doubles` and also the method `String.format()`, see below, which returns a `String` formated using the "C"

10.10.1 Useful String Methods

The `String` object has large range of methods which "act" on the `String`. This is nice example of how Objects and Methods interact. This list in not complete, but it contains the simplest and most frequently used methods, see the full documentation for a complete list:

1. **Length of String**. `int` value returned by

   ```
   stringName.length()
   ```

   Note the `()` are essential since `length` is a Method.

2. **Contents Equality**[3]. `boolean` value returned by

   ```
   string1.equals(string2)
   ```

   if the contents are `string1` and `string2` are identical. There is also a case insensitive version, being

   ```
   string1.equalsIgnorecase(string2)
   ```

   which ignores the difference between upper and lower case letters.

   There are many much more complex comparitor Methods, a few of which are, `compareTo()`, `regionMatches()`, `startsWith()` and `endWith()` which allow ordering of strings, comparisons of sections, and comparison of starts/ends . See full documentation if you need these.

3. **Location of Characters:** Finds the location of a a specified `char` in a String, returning its location so

   ```
   stringName.indexOf('a');
   ```

   returns the location of `char 'a'` in the String, (-1 if it does not exist). There are also variants to start the search at a specified location in the String, and to look for the *last occurrence* of a specified `char`, this being `lastIndexOf(char)`.

4. **Extracting Substrings**: Returns a `String` that is part of a longer `String`. There are two methods with either one or two `int` arguments, these being:

   ```
   stringName.substring(start)
   ```

[3] Do NOT use == it does not do what you expect!.

which returns a `String` starting at the `start` chacater and extending to the *end* of the String, and

```
stringName.substring(start , end)
```

which returns a `String` staring at the `start` character and ending at the `end - 1` character. Both generate at error if the initial string is not long enough.

5. **Concatenation** As well as the + operator the Method

```
string1.concat( string2)
```

returns a **new** String that consists of `string1 + string2`.

6. **Case Conversions:** There are two useful case conversion methods, these being,

`stringName.toLowerCase()` and `stringName.toUpperCase()`

which return a new String with all characters in the `String` converted to Lower Case and Upper Case respectively.

7. **printf style format:** is provided by the static String method

```
String.format(String template, Object, Object ...);
```

which gas *exactly* the same syntax as the `printf()` format scheme convered in the basic-io section, so for exampe can write

```
double x = Math.PI;              // Set x to PI
int i = 1024;
String oString = String.format("Value of i is %d and PI is %.4g",i,x);
```

which will format `i` and `x` into a String using the supplied template so setting `oString` to,

```
"Value of i is 1024 and PI is 3.143"
```

In addition there are a range of more complex String methods which are mainly used in non-numeric applications.

10.11 Arrays of Objects

In addition to the simple *arrays* and *Strings* discussed above, we can form an *array* of **any** Object, for example you can form an *array* of `Graphs`, or and *array* of `Displays`. While this is possible, a better scheme for creating *arrays of objects* is to use the `Vector` or `ArrayList` classes from the `java.util` package which dynamically add and remove objects.

The use of this class is beyond the scope of this course, look it up in books or on-line documentation when you need it.

Examples

The following on-line source examples are available:

- Fill one-dimensional array with squared numbers and sum SquareAdd
- Read in two five element vector and form the scalar product ReadVector
- Multiply two matrix together with for loops (tricky!!!), MatrixMult
- read in a file name in ‘name.suffix’ format and split into ‘name’ and ‘suffix’ FileSplit

What Next?

One more section on *Methods* before the next checkpoint, keep on reading.