Ian Kenny

October 3, 2017

Software Workshop

Lecture 2c

In this lecture

- Introduction to variable scope.
- Arrays.
- Multi-dimensional arrays.
- Nested loops.

Variable scope

In Java, references to a named entity (e.g. a variable) are restricted to those 'parts of the program' where the entity is in *scope*. For example, a variable name may only be used where it is in scope. In the case of a variable, scope can be thought of as the parts of a program where the variable may be referred to and used by name. If a variable is not *in scope* at the point at which the program attempts to access it, the compiler will give an error.

With the types of program we have looked at so far, scope is a very limited and easy to conceive of concept. Let's look at a program.

Variable scope: demo

```java
public class ScopeDemo1 {
    public static void main(String[] args) {

        // this variable is in scope throughout the main method
        int i = 0;

        // so is this one
        int j = 2;

        if (i < j) {
            i += 4; // i is in scope here, so this is fine
        }

        else {
            int k = 5; // k is declared inside the else block
            k *= j;
        }

        System.out.println("i = " + i); // fine
        System.out.println("j = " + j); // fine
        System.out.println("k = " + k); // ummm
    }
}
```

Listing 1 : ScopeDemo1.java

Variable scope

The variable `k` is declared inside the `else` block and is only in scope until *the end of that block*.

The closing brace of the `else` block closes the block hence ends the scope of variables declared inside the block.

This means that the attempt to access `k` in the output statement will cause a compiler error.

```
ian@ian-laptop:/media/usb/sww/SWW2$ javac ScopeDemo1.java
ScopeDemo1.java:21: error: cannot find symbol
System.out.println("i = " + k); // ummm
^
symbol:  variable k
location: class ScopeDemo1
1 error
```

Listing 2 : Terminal commands

Variable scope

Blocks can be declared within other blocks. Indeed, this will always be the case in your programs [1]. If a block ('the outer block') has a block declared inside it ('the inner block') then names declared in the outer block are in scope inside the inner block but not *vice versa*. This is demonstrated in the program on the next slide.

[1] after all, the `main` method block is declared inside the `class` block

Variable scope: demo

```
public class ScopeDemo2 {
    public static void main(String[] args) {

        int i = 0;
        int j = 2;

        if (i < j) {

            int k = 5;

            if (i < k) {

                k++; // accessible here
                int m = 7;
            }

            else {

                k--;
            }

            m++; //oops. Not accessible here
        }
    }
}
```

Listing 3 : ScopeDemo3.java

Variable scope

Where you declare variables defines their scope. You cannot access a variable that is not in scope.

One response to this (in terms of the programs we have seen to date) might be to declare all of your variables at the start of the `main` method. That would mean that those variables can be accessed anywhere inside the `main` method.

This is not considered to be good practice, however. Variables should be declared as close to the point at which they will be used as possible. If they are declared 'early' (e.g. at the start of the `main` method) then that could confuse any programmers reading the code. Where are those variables used? They'd have to hunt for that.

Variable scope

Sometimes you will need to declare a variable ‘early’ because you want to use it in a number of places. That is fine but you should at least consider if you could find a better way of achieving that.

Arrays

It is often the case that you need to store multiple data items of the same type. It is also often inconvenient to declare multiple separate variables for the purpose. Consider the program on the next slide. What problems can you see with it?

Arrays (lack of): demo

```
import java.util.Scanner;
public class ArrayDemo1 {
    public static void main(String[] args) {

        int age1 = 0;
        int age2 = 0;
        int age3 = 0;
        int age4 = 0;

        int index = 0;

        Scanner in = new Scanner(System.in);

        System.out.println("Enter four ages.");

        age1 = in.nextInt();
        age2 = in.nextInt();
        age3 = in.nextInt();
        age4 = in.nextInt();

        System.out.println("Average age is: " + (age1+age2+age3+age4)/4.0f + ".");
    }
}
```

Listing 4 : ArrayDemo1.java

Arrays

The program on the previous slide is bound very tightly to there being always four ages entered. This may be fine. Depending on the application, it may always be the case that exactly four ages will need to be entered. But, it is doubtful that this is really the correct approach.

For the program to work with a different number of ages than four, it needs to be amended. Also, what if the program needed to be amended to deal with 100 ages? Continuing in this fashion is error-prone and tedious, etc. What if it later needs extending to 1000 ages ...?

This is exactly the kind of situation that requires an *array*.

Arrays

An array allows us to easily store multiple items of the same type. For now we are considering only 'primitive arrays', i.e. the arrays built in to the Java language. They offer more support than offered by arrays in C and C++ but they are still fairly basic.

One of the key limitations of built-in arrays is that they are *static*, i.e. the number of elements they may contain is fixed when they are created and cannot be altered.

A key change when using arrays is that each variable in the array does not have an explicit name. Or, rather, its name is simply the name of the array in conjunction with an index. As usual, some code will make this more obvious.

Always keep in mind that the first element of an array is element 0. The final element of the array is element $n - 1$ where n is the number of elements in the array.

Arrays : demo

```
import java.util.Scanner;
public class ArrayDemo1a {
    public static void main(String[] args) {

        // create an array
        int[] values = new int[6];

        // put some values into the array
        values[0] = 3;
        values[2] = 7;
        values[5] = 12;

        // print out the array values
        for (int i = 0; i < 6; i++)
            System.out.print(values[i] + " ");

        System.out.println(); // prints a blank line

        // calculate the sum of the values in the array
        float sum = 0.0f;

        for (int i = 0; i < 6; i++)
            sum = sum + values[i];

        // calculate the average
        float avg = sum/6;

        System.out.println("The average value is " + avg);
    }
}
```

Listing 5 : ArrayDemo1a.java

Arrays : demo

```
import java.util.Scanner;
public class ArrayDemo2 {
    public static void main(String[] args) {

        final int NUM_VALUES = 5;

        int values[] = new int[NUM_VALUES];

        for (int index = 0; index < NUM_VALUES; index++) {
            values[index] = index * 2;
        }

        for (int index = 0; index < NUM_VALUES; index++) {
           System.out.print(values[index] + " ");
        }
   }
}
```

Listing 6 : ArrayDemo2.java

Arrays

The programs on the previous slide demonstrates how an `int` array is created. Arrays of other types are created similarly. Note that the array will have five elements and that this size cannot change unless the value of `NUM_VALUES` is changed in the source code.

The name of each element of the array is simply the name of the array followed by its index. The first element in the array is called `values[0]` and the last element is called `values[4]` (why 4?).

Arrays

It is possible to set the size of an array using a variable. The program on the next slide demonstrates this, and some other things.

Arrays : demo

```
import java.util.Scanner;
public class ArrayDemo3 {
    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        System.out.println("How many ages do you wish to enter?");

        int numAges = in.nextInt();

        int ages[] = new int[numAges];

        System.out.println("Enter the ages.");

        int index = 0;

        do {
            ages[index] = in.nextInt();

        } while (++index < numAges);

        System.out.println("You entered the following ages:");

        float sum = 0;

        for (int i = 0; i < numAges; i++) {
            System.out.print(ages[i] + " ");
            sum = sum + ages[i];
        }

        System.out.println("\nAverage age is: " + sum/numAges + ".");
    }
}
```

Listing 7 : ArrayDemo3.java

Arrays

Arrays can be declared using some alternative methods.

```
public class ArrayDemo4 {
    public static void main(String[] args) {

        int[] a1 = new int [20];

        final int NUM_VALUES = 5;
        int a2[] = new int[NUM_VALUES];

        int num = 4;
        int a3[] = new int [num];

        // note that the size of the array does not
        // need to be explicitly stated if an
        // initialiser list is used
        int a4[] = {1, 4, 6, 4, 2, 8, 9};
    }
}
```

Listing 8 : ArrayDemo4.java

Array length

Unlike C and C++, the Java built-in arrays have a property called `length` that allows you to find the number of elements in the array. The program on the next slide demonstrates this. Can you see how the program has changed?

In this case the change offers little, but this would be very useful where you do not have access to the variable that was used to size the array (which could be out of scope or otherwise not accessible to you).

Arrays : demo

```
import java.util.Scanner;
public class ArrayDemo5 {
    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        System.out.println("How many ages do you wish to enter?");

        int numAges = in.nextInt();

        int ages[] = new int[numAges];

        System.out.println("Enter the ages.");

        int index = 0;

        do {
            ages[index] = in.nextInt();

        } while (++index < ages.length);

        System.out.println("You entered the following ages:");

        float sum = 0;

        for (int i = 0; i < ages.length; i++) {
            System.out.print(ages[i] + " ");
            sum = sum + ages[i];
        }

        System.out.println("\nAverage age is: " + sum/ages.length + ".");
    }
}
```

Listing 9 : ArrayDemo5.java

Array: bounds checking

Java automatically checks at runtime if an array access would be *out of bounds*. This means an attempt to use an index that is less than zero or greater than the array's length - 1. If such an access is attempted, an exception is thrown by the runtime system. For example, if you create an array of length 5, then you cannot access any element that is less than zero or greater than 4 (without causing an error). Incidentally, the same rule applies to strings. If you attempt to use `charAt()` with a negative parameter or a parameter greater than 'string length - 1' you will get an error.

Multi-dimensional arrays

So far we have looked at arrays that contain a simple primitive type. Arrays can contain more complex types (as we will see). Arrays can also contain arrays. Arrays containing arrays are sometimes called *multi-dimensional arrays*.

Each element of a mult-dimensional array is an array. The arrays contained in the array may also contain arrays!

Thinking geometrically, the arrays seen so far are *one-dimensional* arrays. They are accessed in a purely 'linear' fashion. We can only move forward or backward (left and right) along the array (in one dimension).

However, It is common to use *two-dimensional arrays*, for example. A two-dimensional (2D) array can be thought of as a 'grid' or 'matrix' of values, having *rows* and *columns*. With such an array we can move forwards and backwards along the rows, up and down the columns, or a combination of both. Again, indices start at zero.

Consider the programs on the next slide.

Arrays of arrays: demo

```
public class ArrayDemo6 {
    public static void main(String[] args) {

        /* declares an array that has two elements. Each
            element of this array is an array of three
            elements.

        Visualisation:

        a[0] -> [0][1][2]
        a[1] -> [0][1][2]

        */

        int[][] a = new int[2][3];

        a[0][0] = 2;
        a[1][2] = 7;
        a[2][0] = 4; // error: out of bounds

    }
}
```

Listing 10 : ArrayDemo6.java

Note that the first index gives the 'row' of the matrix and the second the 'column'.

Arrays of arrays : demo

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

        int[][] sudokuBoard = new int[9][9];

        // top-left element
        sudokuBoard[0][0] = 0;

        // bottom-right element
        sudokuBoard[8][8] = 0;
    }
}
```

Listing 11 : Sudoku1.java

Nested loops

It is convenient to access multi-dimensional arrays using loops (as with one-dimensional arrays) but it is more natural to access them using *nested loops*.

A nested loop is a loop that has one or more other loops declared inside it. If, for example, a nested loop has two loops - the outer loop and the inner loop - the inner loop executes **fully** for every **single** iteration of the outer loop.

Consider the revised Sudoku program on the next slide. In this program, the Sudoku boad is created and then all elements are initialised to zero using a nested loop.

Nested loop : demo

```
public class Sudoku2 {
    public static void main(String[] args) {

        final int N_ELEMENTS = 9;

        int[][] sudokuBoard = new
            int[N_ELEMENTS][N_ELEMENTS];

        /* a nested loop to initialise the sudoku board to
            all zero values */

        // accesses each row of the board
        for (int i = 0; i < N_ELEMENTS; i++) {

            // accesses each column
            for (int j = 0; j < N_ELEMENTS; j++) {

                sudokuBoard[i][j] = 0;
            }
        }
    }
}
```

Listing 12 : Sudoku2.java

The inner loop (index `j`) iterates N_ELEMENT times for each single iteration of the outer loop (index `i`).

Arrays of arrays : demo

What will the output be?

```
public class ArrayDemo7 {

    public static void main(String[] args) {

        final int ARRAY_DIM = 4;
        int[][] values = new int[ARRAY_DIM][ARRAY_DIM];

        for (int row= 0; row < ARRAY_DIM; row++) {
            for (int column = 0; column < ARRAY_DIM; column++) {
                values[row][column] = row + column;
            }
        }

        System.out.println("Index[1][1] = "+ values[1][1]);
        System.out.println("Index[2][3] = "+ values[2][3]);
        System.out.println("Index[0][3] = "+ values[0][3]);

    }
}
```

Array of arrays: demo

```
$ javac ArrayDemo7.java
$ java ArrayDemo7
Index[1][1] = 2
Index[2][3] = 5
Index[0][3] = 3
```

Listing 13 : Terminal commands

Bubblesort

A nested loop can be used to *sort* an array into a specific order. For example, an array of integers can be sorted into an ascending or descending order. Consider the program on the next slide which implements a basic (inefficient) sort called *bubblesort* which sorts an array into ascending order.

This algorithm is called *bubblesort* because in order to sort the array it 'bubbles' values 'up' the array, i.e. moves them along until they are in the correct position.

Bubblesort

```
public class Bubblesort {
    public static void main(String[] args) {
        /* the numbers to sort into ascending order */
        int[] values = {2,5,1,9,7,7};
        boolean swapped = true;

        while (swapped == true) {
            swapped = false;
            for (int i = 1; i < values.length; i++) {
                if (values[i-1] > values[i]) {
                    int temp = values[i];
                    values[i] = values[i-1];
                    values[i-1] = temp;
                    swapped = true;
                }
            }
        }

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

Bubblesort

See if you can follow how this algorithm works. If necessary, write out the steps it is taking on a piece of paper. Use a smaller array as an example (in your paper version). For example, an array with only three or four elements.