



Introduction to Programming

12. Arrays – part 3

1



An aside on programming errors!

n Last week's lecture finished with the following two slides...

2



An example application

```
public static void main(String[] args) {  
    int[] primesLessThan20 =  
        {11,7,3,17,5,13,2};  
    int[] minAndMax =  
        findMinAndMax(primesLessThan20);  
    TextIO.putln("The smallest prime is " +  
        minAndMax[0]);  
    TextIO.putln("The largest prime less " +  
        " than 20 is " + minAndMax[1]);  
}
```

3




Application output

n The application outputs:

The smallest prime is 2

The largest prime less than 20 is 17

4



There was a significant error in this program's output

n What is it?

5



Testing and errors

- n What is the largest prime that is less than 20?
 - n Hint: it is not 17
- n Lesson: programs might compile with no problems and appear to behave correctly and yet be incorrect
 - n Always test your program by devising test cases, working out what the correct output/result should be, and comparing it with what the program actually outputs

6



That error...

- n There is nothing wrong with the logic of the program, the problem is with its data – the array should include the largest prime less than 20 which (you'll have worked out) is 19
- n Most programs need the developing team to have knowledge not just of programming but also of the application domain that the program is intended to be used in
- n Well done if you pointed out the error in your class last week!

7



Now, back to arrays....

8

Arrays so far

- n Arrays in Java are *objects* – an array variable contains a reference to an array object (if its value isn't `null`)
 - n You need to both declare the array variable *and* create the array object
- n An array object is a numbered list of elements of the same type
 - n Each element can be thought of as a variable – so you can also think of an array as being a numbered list of variables of the same type
- n The elements can be primitive types (e.g. `int`, `double`, `boolean`, `char`) or objects (e.g. `String`)

9

Arrays so far - continued

- n Objects need to be explicitly created so if the elements of an array are objects, you need to remember that when the array object is created its elements are all initially `null` and you need to also create those objects
- n If the elements in an array are themselves array objects you have a *multidimensional array*
 - n You can think of the first dimension as a set of rows, each of which is an array (this is what you index with the first number)
 - n The second dimension (indexed with the second number) is a set of columns each of which is an array element (for a three dimensional array this element would also be an array, for a two dimensional array it would be a primitive type or an object such as a `String`)

10

Arrays so far - continued

- n The great thing about an array, as opposed to many other data structures that include a collection of elements, is that you can use the index number of an element to go *directly* to it without having to go through any of the other elements of the collection
 - n Arrays are direct access structures
 - n Accessing an array element by its index is fast!
- n This was one of the main points of explaining that array elements are stored next to each other (consecutively) in memory – the virtual machine can work out where each element is if you have its index number

11

Iterating in reverse over an array

- n It was an exercise in the lab last week to display the elements of an array in reverse order
- n Simplest way is to use a for loop
 - n set the loop counter (used to index the array) to the array length-1 (index of the last element in the array)
 - n subtract one from the counter each time round the loop
 - n continue till the counter reaches 0 (the index of the first element in the array, remember to include 0)
- n Example is on the next slide

12

Iterating over an array in reverse

```
int[] primes = {2,3,5,7,11,13,17,19};
```

Start at the last element

>= as we are going down and want to include 0

Finish at the first element

```
for (int i=primes.length-1; i>=0; i--) {  
    TextIO.putln(primes[i] + " is prime");  
}
```

13

Application output

n The application outputs:

```
19 is prime  
17 is prime  
13 is prime  
11 is prime  
7 is prime  
5 is prime  
3 is prime  
2 is prime
```

14

The enhanced for loop

n We have met that there is an enhanced for loop (aka a foreach loop) that can also be used to iterate over an array

- n Avoids having to declare and increment a loop counter and explicitly index the array
- n But cannot be used to
 - n update the array elements
 - n iterate over only part of the array
 - n iterate over the array in reverse order

15

The enhanced for loop cont'd

n You cannot use a single foreach loop to iterate over a multidimensional array's elements, but you can nest these loops, just as you would a normal for loop

n Use a pair of nested foreach loops to iterate over a 2D array

```
double[][] matrix = {                // 2x3 array  
    {1.0,0.0,0.0},                    // matrix[0]  
    {0.0,1.0,0.0}, };                 // matrix[1]  
for (double[] row : matrix) {         // rows  
    for (double element : row) {     // columns  
        TextIO.putln(element);  
    }  
}
```

16



Arrays and Methods

n Var-arg parameters

- n Before Java 5, every method had a fixed number of parameters, and every time the method was called you had to supply an argument for each of those parameters
- n Most methods are still like this
- n After Java 5, it became possible to write a method that has a variable number of parameters – that is, it is possible for different calls to the method to supply a different number of arguments

17



Var-arg methods (see section 7.1.2 of the textbook)

- n This is what the first line of a var-arg method declaration looks like:

```
public static double average(double... numbers){
```

- n You call the method just like any other, except that different calls can have different numbers of arguments:

```
average(1,4,9,16);      average(3.14,2.17);  
average(0.375);         average();
```

18



Var-arg methods continued

- n When the method is called, the list of arguments is automatically copied into an array whose name is whatever the parameter name is
- n This means it is also legal to call a var-arg method with an array argument:

```
double[] list = {0.0,1.0,2.0,3.0};  
double d = average(list);  
TextIO.putln(d); // outputs 1.5
```

19



Var-arg methods continued

- n The code for this method given in the book is:

```
public static double average( double... numbers ) {  
    // Inside this method, numbers is of type double[].  
    double sum; // The sum of all the actual parameters.  
    double average; // The average of the actual parameters.  
    sum = 0;  
    for (int i = 0; i < numbers.length; i++) {  
        // Add one of the actual parameters to the sum.  
        sum = sum + numbers[i];  
    }  
    average = sum / numbers.length;  
    return average;  
}
```

20

Var-arg methods continued

- n Or you could use a foreach loop as the method does not update the array:

```
public static double average(double... numbers){
    double sum = 0.0; // sum of the parameters
    for (double current : numbers) {
        // add next parameter to the sum
        sum += current;
    }
    return sum/numbers.length;
}
```

21

Var-arg methods

- n The book calls these methods, *variable arity methods*
- n A method can only have one var-arg parameter
- n If the method has other parameters, the var-arg parameter must come last
- n We have already met one var-arg method: `TextIO.printf()`

22

The ArrayList class

- n Java provides a class, `ArrayList`, to represent *dynamic arrays*
- n A dynamic array does not have a fixed length, its size increases as you add things to it and decreases as you remove things from it
- n To use an `ArrayList` in your program you need to import it from `java.util`

23

The ArrayList class continued

- n You cannot use square brackets `[]` with an `ArrayList`
- n To find the size of an `ArrayList` you call its `size()` method
- n To add an item to the end of an `ArrayList` you call its `add()` method
- n To retrieve the item at index position *i* in an `ArrayList`, you call its `get()` method and supply the value of *i* (must be between 0 and `size()-1`)

24

The ArrayList class continued

- n To change the value of the item at index position *i* in an ArrayList, you call its *set()* method and supply the new item value and the value of *i*
- n To remove the item at index position *i* in an ArrayList, you call its *remove()* method and supply the value of *i* (must be between 0 and *size()-1*)
- n To find the index position of an item in an ArrayList you call its *indexOf()* method and supply the value of the item you are looking for

25

The ArrayList class continued

- n An ArrayList holds a collection of *objects*
- n If you want an ArrayList to hold values of a primitive type, you can use the *wrapper class* for that type
- n The wrapper classes allow you to automatically convert a primitive type value into a reference type value:

Integer for **int**

Character for **char**

Double for **double**

Boolean for **boolean**

Long for **long**

Short for **short**

Byte for **byte**

Float for **float**

26

The Wrapper Classes (see 7.3.2 of textbook)

- n Each primitive type has a wrapper class that allows you to treat a value of the primitive type as an object
- n The primitive type value is automatically *boxed* in an object (autoboxing)
- n Once boxed, it can be automatically unboxed

```
Integer myInt = 7; // box 7
```

```
int number = myInt*2; // unbox 7
```

27

Declaring an ArrayList

- n To declare and create an ArrayList whose elements are Strings you would write:

```
ArrayList<String> list =  
    new ArrayList<String>();
```

Note the angle brackets

- n To declare and create an ArrayList whose elements are ints you can use int's wrapper class and write:

```
ArrayList<Integer> list =  
    new ArrayList<Integer>();
```

28

Example program (section 7.3.3 of the textbook)

```
import java.util.ArrayList;

/**
 * Reads a list of non-zero numbers from the user,
 * then prints out the input numbers in the reverse
 * of the order in which they were entered.
 * There is no limit on the number of inputs.
 */
public class ReverseWithArrayList {
    public static void main(String[] args) {
        ArrayList<Integer> list;
        list = new ArrayList<Integer>();
```

29

Example program continued

```
// Read the numbers from the keyboard
TextIO.put("Enter some non-zero integers.");
TextIO.putln(" Enter 0 to end.");
while (true) {
    TextIO.put("? ");
    int number = TextIO.getlnInt();
    if (number == 0)
        break; // stop when the number is 0
    list.add(number); // add to end of list
}
```

30

Example program continued

```
TextIO.putln();
TextIO.putln("Your numbers in reverse are:");

// Now iterate over the ArrayList in reverse
for (int i = list.size() - 1; i >= 0; i--) {
    TextIO.printf("%10d\n", list.get(i));
}
```

Display the integer in a field that is 10 characters wide and take a new line afterwards: see Lab 07

31

ArrayList and the foreach loop

- n You can use a foreach loop with an ArrayList

```
ArrayList<String> myList =
    new ArrayList<String>();
for (String s : myList) {
    TextIO.putln(s);
}
```

32



foreach loop and unboxing

- n If the ArrayList elements are objects of a wrapper class you can use the primitive type in the foreach loop, as the objects will be automatically unboxed

```
ArrayList<Integer> myIntList =  
    new ArrayList<Integer>();  
for (int i : myIntList) {  
    TextIO.putln(i);  
}
```

33



Summary

- n Today we have looked at
 - n Using a pair of nested foreach loops to iterate over a two-dimensional array
 - n Variable arity methods, whose arguments are automatically added to an array for the method to process so that the method can be called with different numbers of arguments on different calls (var-args)
 - n The wrapper classes for the primitive types
 - n The java.util.ArrayList class as a dynamic array type

34



Lecture source code

- n Try pasting the code for the average() method into a class and call it a few times from your main() method with different numbers of arguments
- n The example ReverseWithArrayList program is on Moodle in the subpage for today's class – download this and run it

35



Any questions?

36