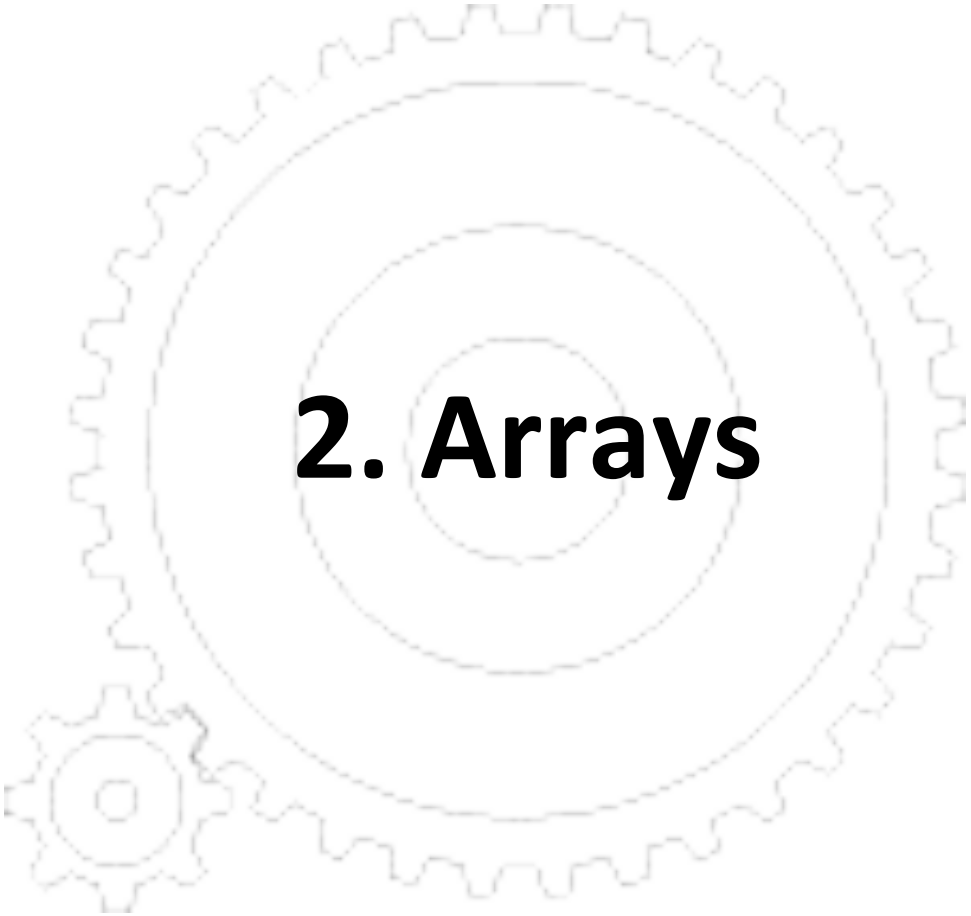
G51OOP



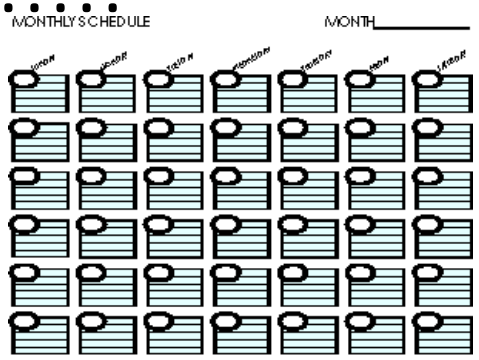
2. Arrays

Object Oriented Programming

Colin Higgins

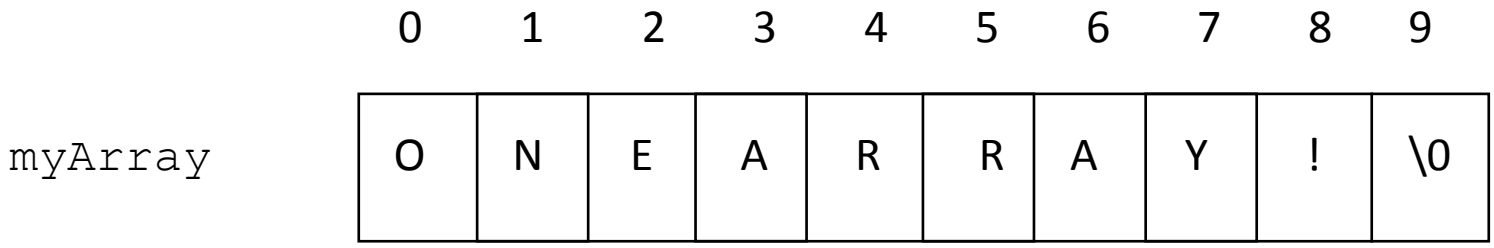
Motivation

- So far we have declared variables one at a time.
- We often need to group variables in a program
- Arrays are a technique for :
  - Grouping many variables of the same type and
  - Being able to treat the whole collection as a single entity for appropriate operations.
- NOTE: you will probably use “container” classes in more complex programs!!!!!!

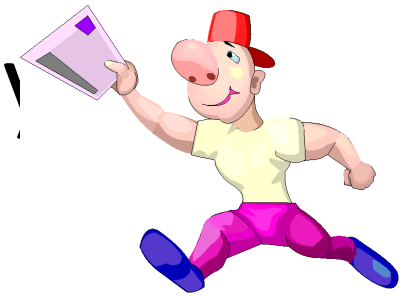


Arrays

- An *array* is an ordered list of values of the same type
- Each value has a numeric *index*
- An array of size N is indexed from 0 to N-1
- The following array of characters has a size of 10 and is indexed from 0 to 9



Declaration of array



- Syntax : `type[ ] identifier` OR `type identifier[ ]`
- So, the previous examples could be declared as :

```
float[] annualRain;    // A float array
char[] text;           // A char array
int[] studNumbers;     // An int array
```

- Or in a C++ declaration style :

```
float annualRain[];    // A float array
char text[];           // A char array
int studNumbers[];     // An int array
```

Allocating memory for arrays

- We need to allocate space for the elements of the array.

```
 // allocate 90 floats
 static final int RAIN_SIZE = 90;
annualRain = new float[RAIN_SIZE];

// allocate 10000 chars
static final int CHAR_SIZE = 10000;
text = new char[CHAR_SIZE];

// allocate 100 ints
static final int MAX_STUDENTS = 100;
studNumbers = new int[MAX_STUDENTS];
```

Declaration and Allocation of arrays

- Declaration and allocation can be done in a single statement.

```
float[] annualRain = new float[RAIN_SIZE];

char[] text = new char[CHAR_SIZE];

int[] studNumbers = new int[MAX_STUDENTS];
byte[] byteBuffer = new byte[BUFFER_SIZE];
Money[] savings = new Money[ACCOUNT_COUNT];
```

- In Java, the size of the array can be defined at runtime.

```
int size = G51OOPInput.readInt();
int arr[] = new int[size];
```

Accessing array elements

Eg

```
static final int RAIN_SIZE = 90;
float[] annualRain = new float[RAIN_SIZE];
```

The subscripts run from 0 to 89 inclusive; giving us the required number (90) of variables. To access the 23rd of these variables, we would put the subscript in square brackets after the array identifier, and use :

```
... annualRain[22] …
```

The subscript can be any integer expression.

To store a value in this variable we may use :

```
annualRain[22] = 42.0;  or perhaps
annualRain[22] = G51OOPInput.readFloat();
```

and to use the value stored there :

```
i = 22;
if ( annualRain[i] > MINIMUM ) {
    total = total + annualRain[i];
  }
```

Array Bounds

- **Notes :** Arrays are a sort of "object" (details later) and as such there is a length field available. To access this field use the dot notation as follows.

```
final int SIZE = 1024;
int[] a = new int[SIZE];
int b;
b = a.length;
// b now has the value 1024.
```

- This length field is a final field and cannot be changed. You **cannot** attempt to reset the length. The following is NOT allowed:

```
a.length = 7;
```

- In Java, unlike C/C++, array bounds are checked at run time, if an error occurs an ArrayIndexOutOfBoundsException is thrown.

ArrayIndexOutOfBounds Exception

```
public class ExceptionExample {
   public static int ARRAY_SIZE = 10;
   public static void main (String[] argv) {
    int [] numArray;                        // declaration
    numArray = new int[ARRAY_SIZE];         // creation
    try {
           for (int x=0; x<numArray.length +1; x++) {
            numArray[x] = x * 2;
            System.out.print(numArray[x] + ":");
            }
    }
    catch (ArrayIndexOutOfBoundsException ex) {
       System.out.println("Array Exception caught!!");
    }
    finally {
        System.out.println("Reached finally clause");
    }
   }
}
```

Causes the Exception

Destroying Arrays

- When you no longer need an array you can simply ignore it – the garbage collector will tidy it up for you (see objects later).
- For example :

```
static final int MAX_ELEMENTS = 10;
static final float NEW_MAX = 20;
double[] myDoubles;
myDoubles = new double[MAX_ELEMENTS];
// do something with 10 doubles
...
myDoubles = new double[NEW_MAX];
// do something with 20 different doubles
...
```

Arrays used in loops

- The usefulness of arrays reveals itself when we refer to each element of the array not by a specific constant subscript, but access all elements in turn or choose a particular element dynamically. To access the variables in turn, for example to read 90 values from the data into the 90 locations, we could use

```
For(int year = 0; year < annualRain.length; year++) {
   // Variable "year" goes from 0 to 89
   annualRain[year] = G51OOPInput.readFloat();
} // for year
```

- The 90 values would have to appear in the data stream in the correct order, separated by "white space", i.e. spaces, tabs or newlines, depending on the reading format.
- The typical Java loop, starting at zero, and limited by "counter strictly less than number of elements". This gives us the range of values 0, ..., 89 if there are 90 elements. There is NO element with subscript 90.

Loops - Finding the Average Value Example


```
float total = 0.0f;

For (int year = 0; year < annualRain.length; year++) {
   annualRain[year] = G51OOPInput.readFloat();
}

for (int year = 0; year < annualRain.length; year++) {
   // Each array element is a "float"
   total = total + annualRain[year];
}

System.out.println("Total : " + total );
System.out.println("Average : " + total/
  annualRain.length);
```

Initialised arrays

If you wish the values of an array to be initialised then you could write :

```
int[] primes = { 1, 2, 3, 5, 7, 11 };
```

with the initial values comma-separated, between curly braces. You do not need to give

an explicit value for the length of the array, because the compiler can count how many

elements you have declared! The above statement would set up an array of six elements

starting at suffix zero :

```
primes[0] = 1
primes[1] = 2
primes[2] = 3
primes[3] = 5
primes[4] = 7
primes[5] = 11
```

Initialised Arrays

```
char[] letter_grades = {'A', 'B', 'C', 'D', 'F'};
double[] percentages = { 1.50, 1.35, 1.15, 1.00, 0.85, 0.60 };
boolean[] truthTable = { true, false, false, false};
```

- An initialiser list can only be used in the declaration of an array
- the `new` operator is not used

Declaring Two-dimensional a

```
static final int N_YEARS = 90;
static final int N_MONTHS = 12;
float[][] monthRain = new float[N_YEARS][N_MONTHS];

static final int CLASS_SIZE = 90;
static final int SUBJECTS = 12;
static final int STUDENT_NUMBERS = 200;
static final int MAX_MODULES = 30;

int[][] IQ = new int[CLASS_SIZE][SUBJECTS];
int[][] mark = new int[STUDENT_NUMBERS][MAX_MODULES];
char[][] names = new char[STUDENT_NUMBERS][MAX_MODULES];
```

Two-dimensional arrays

- To read rainfall data in, we might use :

```
// Read in all the 12 * 90 values

For (int year = 0; year < monthRain.length; year++) {

  for (int month=0; month < monthRain[year].length; month++) {
       monthRain[year][month] = G51OOPInput.readFloat();
  } // Month loop

} // Year loop
```

Accessing the elements

- To calculate the annual totals for each of the 90 years we might write :

```
// Calculate the year totals
Static final CENTURY = 1900;
float total;
For (int year = 0; year < monthRain.length; year++)
{
   total = 0;
   for (int month= 0; month < monthRain[year].length;month++){
       total = total + monthRain[year][month];
   } // for month
   annualRain[year] = total;
   System.out.println("year " + year + CENTURY + " rain " +
              total);
} // for year
```

Initialising two-dimensional

- You could write for example :
- ```
  int[][] table = { { 1, 2, 3, 4, 5 },
   { 5, 4, 3, 2, 1 }, { 1, 3, 5, 3, 1 }};
  ```

- It should be noted that Java implements multidimensional arrays as arrays-of-arrays. This means that multidimensional arrays need not be rectangular!

- ```
  Hence  short[][] triangle = { { 1 }, { 2, 3 },
  { 4, 5, 6 }, { 7, 8, 9, 10 } };
  ```

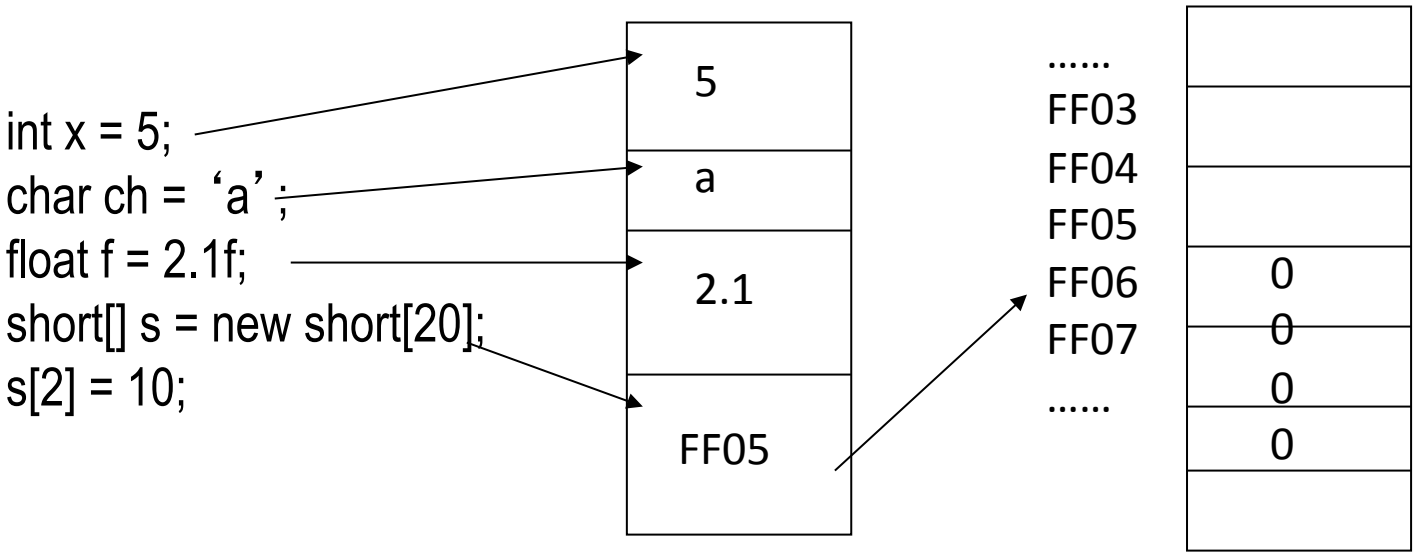
Higher dimensional arrays

- You can, of course, declare and use 3-dimensional, 4-dimensional and higher dimensioned arrays by extending the above notation, although with object oriented programming this is usually avoided.

```
// In three dimensions :
Stsic final int SZIE = 10;
float[][][] mass = new float[SIZE][SIZE][SIZE];
for ( int x = 0; x < mass.length; x++ ) {
   for ( int y = 0; y < mass[x].length; y++ ) {
       for ( int z = 0; z < mass[x][y].length; z++ ) {
          if ( mass[x][y][z] >= min ) {
             ...
          } // end if
       } // Z loop
   } // Y loop
} // X loop
```

Stack and Heap

- When the Virtual Machine (java) executes a program it uses two memory areas : The Stack and the Heap.
- All contents of the 8 datatype variables are stored in the stack area.
- All contents of arrays and objects are stored in the heap area. Their position in the heap is kept as an address in the stack area.



Arrays as method parameters

- Arrays can be passed as method parameters.
- Arrays are passed by reference

```
static final int SMALL = 10;
Static final int BIG = 1000;
Public static void maon(String argv[]) {
     int[] smallArray = new int[SMALL];
     int[] bigArray = new int[BIG];
     myMethod(smallArray);
     myMethod(bigArray);
}

public static void myMethod(int[] anArray) {
     System.out.format("size = %6d%n", anArray.length);
}
```

Interactive development example

- Knights tour of a chess board
- Use **Warnsdorff's rule**