TEESSIDE UNIVERSITY - SCHOOL OF COMPUTING, ENGINEERING AND DIGITAL TECHNOLOGIES
OBJECT ORIENTED PROGRAMMING

Last week we looked at a program to calculate the gross pay, net pay and tax owed for an employee. We then modified the code to use a loop to repeatedly enter information and calculate the pay and tax for several employees.

What if we wanted to store all the employee data that we entered? How would we do this?

A possible approach would be to use something called an **Array**. This will be our starting point, hopefully you will be able to see the benefits of using an Array to store similar data in an aggregate data structure (they're grouped together and occupy contiguous locations in memory).

Think of an Array like a spreadsheet or a table in Word that contains two rows, and some previously defined fixed-length:

INDEX	0	1	2	3	4
DATA	8	4	99	6	1

You have no control of the top-row but do take note of the numbers. The top row are the *indices*. You do have control of the data stored in the bottom row. The position inside the spreadsheet or table where you will find the data. As you can see, we're using the table above to store numbers only. In Java, an Array can only store data of the same type. We'll assume that our table is storing *integer* values as there are no decimal values present. In Java, we'd create a fixed-length Array of integer values like this:

```
// version #1
// allocate an array of 5 integer elements
int numberOfElements = 5;
int[] dailyChangeInSalesOrders = new int[numberOfElements];

// now assign their values:
//
dailyChangeInSalesOrders[0] = 8;
dailyChangeInSalesOrders[1] = 4;
dailyChangeInSalesOrders[2] = 99;
dailyChangeInSalesOrders[3] = 6;
dailyChangeInSalesOrders[4] = 1;


// version #2
// allocate and initialize an array of 5 elements.
//
int[] dailyChangeInSalesOrders = { 8, 4, 99, 6, 1 };
```

Both examples above create a fixed size array of 5 integer elements. Version 1 is better when the number of elements is unknown at the time of writing, for example reading data in from a file or asking the user how many elements they want to create. Note the use of the `[]` on both left and right side of the assignment operator. We could have explicitly stated the size of both sides, but this is neither usual nor helpful (if the size changed, you'd have to change it in two places).

The second version is better when there are know default values, and the number of elements is known at compile time.

Because the data in Arrays are conveniently grouped together and in contiguous memory locations we can then use loops to *iterate* over them, typically a for loop. As previously mentioned there are two types of for loop, we'll start by using the one that you've already used:

```
for(int curIndex = 0;
    curIndex < dailyChangeInSalesOrders.length;
    curIndex++) {
    System.out.println(
        "The daily change in sales orders for day " +
        curIndex +
        " is " +
        dailyChangeInSalesOrders[curIndex]);
}
```

Here we're using the loop variable '**`curIndex`**' to read an element from the array and print its value - the **`[]`** operator is known as the ***subscript*** operator or more informally the '*square brackets*' operator.

We can apply all the same operations (e.g. Arithmetic) to the elements in the array as you would if they were individual variables:

```
int anInt = 10;

anInt += 20;

System.out.println("The value of anInt " + anInt);

dailyChangeInSalesOrders[4] += 99;

System.out.println(
    "The value of dailyChangeInSalesOrders[4] " +
    dailyChangeInSalesOrders[4]);
```

However, the benefit to them being in an array is that you can very easily apply to the same operations to a collection of values using a loop. Observe the use of the *subscript* operator.

Arrays and data types

In the above example we have limited ourselves to an array holding integer values. In Java, arrays (and later collection types from the standard library) can only hold data of the same ***type***. In other words, you can't have one array that can store a mixture of numeric types, strings, and so on. However, you can create arrays of any available data type, for example:

```
double[] arrayOfDoubles = new double[10];
String[] myFriendsNickNames = new String[1000];
char[] grades = new char[moduleCount]; // assume moduleCount is
a value variable
```

Life without Arrays (Bad times)

Consider not using a loop and individual variables:

```
int mondayChangeInSales = 8;
int tuesdayChangeInSales = 4;
int wednesdayChangeInSales = 99;
```

```
    int thursdayChangeInSales = 6;
    int fridayChangeInSales = 1;
```

Now to display this information:

```
    System.out.println("The value of mondayChangeInSales " +
    mondayChangeInSales);
    System.out.println("The value of tuesdayChangeInSales " +
    tuesdayChangeInSales);
    // ... Bored now ...
    System.out.println("The value of fridayChangeInSales " +
    fridayChangeInSales);
```

Now let's add some code to calculate the mean average:

```
int averageChangeInSales =   mondayChangeInSales +
                             tuesdayChangeInSales +
                             wednesdayChangeInSales +
                             thursdayChangeInSales +
                             fridayChangeInSales / 5;
```

Now, notwithstanding the human error introduced by the failure to apply BODMAS/BIDMAS/PEDMAS/OtherMAS, calculating the average is a chore.

Later on we change the sales reporting to include Saturday and Sunday, now we have to create 2 additional variables and add code to process the each variable independently. We have to update the arithmetic to now include Saturday and Sunday, and ensure that we're now dividing by 7 instead of 5. Lots of opportunity for mistakes to be made by the programmer.

Now imagine that you're doing something similar for the entire month or year. This is not a very flexible or desirable approach.

Life with Arrays (Good times)

So for completeness and comparison, lets now see the code for calculating the mean average of the data stored in the array declared above:

```
    int meanAverage = 0;
    for(int index = 0;
        index < dailyChangeInSalesOrders.length;
        index++) {
       meanAverage += dailyChangeInSalesOrders[index];
    }
    meanAverage /= dailyChangeInSalesOrders.length;
```

And using the ***for-in*** style loop (which we've not yet encountered), we can further reduce the verbosity and clutter:

```
    int meanAverage = 0;
    for(int value : dailyChangeInSalesOrders) {
        meanAverage += value;
    }
    meanAverage /= dailyChangeInSalesOrders.length;
```

Why do Array indices start at 0 (zero)?

Why does the first element have the index 0 is a common cause of bewilderment with new programmers! The reason is for it is historically technical and consequently a convention. Some programming language have used 1 (one) as the index of the first element of an array, but on the whole most mainstream languages use 0 (zero) as the index of the first element.

Ok, so why?

Every byte in memory has a unique memory location. When your program is executed, the operating system allocates it an address space. Memory will be allocate for the application code itself, some for any read-only, static data, and some for dynamically allocated data known as ***the heap***. This is a very simplified breakdown, but sufficient for our needs.

The smallest addressable unit of computer's memory is a single byte. Every byte of the computer's memory will have a unique memory address, starting from 0 all the way up to whatever the maximum capacity of the computer's memory. Below is a table that represents a small portion of computer memory. Typically, memory addresses are reported using base-16 (hexadecimal) however for convenience we will use base-10 (decimal) - we will examine the memory from an arbitrary point, but again for convenience we will use 1000.

Address	1000	1001	1002	1003	1004	1005	1006	1007
Data								

Now, each byte of data stores 8-bits of binary - simply looking at the data in memory wouldn't be very informative, it is just lots of 0s and 1s!

Let's consider a variable of type int. An int occupies 4-bytes of memory. In the above table, if our integer variable was stored from memory address 1000, then bytes 1000 through to 1003 would be used for storing our 4-byte integer.

If you have an array of 5 int, then the 5 x 4 bytes of the memory available for your application will be allocated. The integers will be nicely lined up in contiguous memory locations, like the world's most boring and easiest Tetris game.

Assuming that the first integer in our array was stored at 1000, the second element would be located at 1004, the third at 1008. If you look at this carefully, there is a pattern!

```
Starting memory location = 1000
Size of an integer = 4-bytes.
Index of the 1st element = 0
```

So the offset from the first memory location used by the array is:

```
Memory offset = Index of element * 4-bytes

Location of data = Starting memory location + Memory offset.
```

For example, the 4th element would have index 3:

```
Data location = 1000 + 3 * 4-bytes
```

```
                  = 1000 + 12
                  = 1012
```

Another example, the 1st element with index 0:

```
    Data location = 1000 + 0 * 4-bytes
                  = 1000 + 0
                  = 1000
```

If array indexing started a 1 instead of 0, then an additional calculation step would be required, i.e. Subtract 4-bytes to get to the correct location in memory for every element. For simple programs and/or those not handling much data the performance impact of this addition arithmetic step would be negligible, but in more complex applications this would add overhead that would become noticeable.

So, whilst it is not necessarily intuitive for us humans that array indices start at 0, for computers it is appropriate and more efficient.