**Institute of Technology Tralee**

**Computing Department**

**Introduction to Programming**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Practical 7 – Looping with while**

We began our examination of **while** loops last time out and, in particular, we looked at one application of such a loop – in a **data-sentinel controlled** environment. This time we will look at another application scenario for while – for **counter-controlled** iteration.

**Counter-controlled while Loops**

This is the second possible application of a while loop we shall consider. In this case, the software developer (you!) **knows exactly how many times a process will repeat** and so this number can be ‘built in’ to the workings of the while loop at compile-time. This is very useful, especially for systems which have some process that repeats a specific number of times. Examples of such devices include a child’s amusement toy like ‘Postman Pat’ which will move a specific number of times when the required fare has been supplied. Another example would be a passport photograph machine. Here again, provided that the necessary coins have been supplied, the device’s camera will take a certain number of pictures.

The important difference between this type of while loop and the data-sentinel controlled version is that, with counter-controlled repetition, much of the ‘dynamism’ associated with data-sentinel loops is lost and the **end-user has no control over the number of executions of the loop** - it is preordained.

**The General Form of a Counter-controlled while Loop**

Similar to data-sentinel controlled while loops, it is possible to generalise the appearance of a counter-controlled while loop in pseudocode terms as follows:

*counter variable*=*initial value*;

*while***(***counter variable* **<** *final value***)**

*statement1;*

:

:

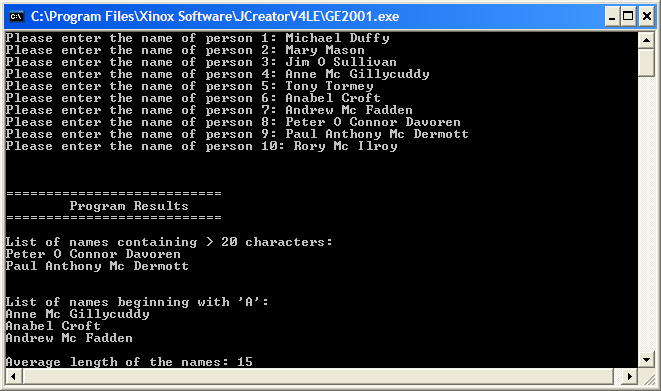
*statementn;*

*counter variable*=*counter variable+increment*;

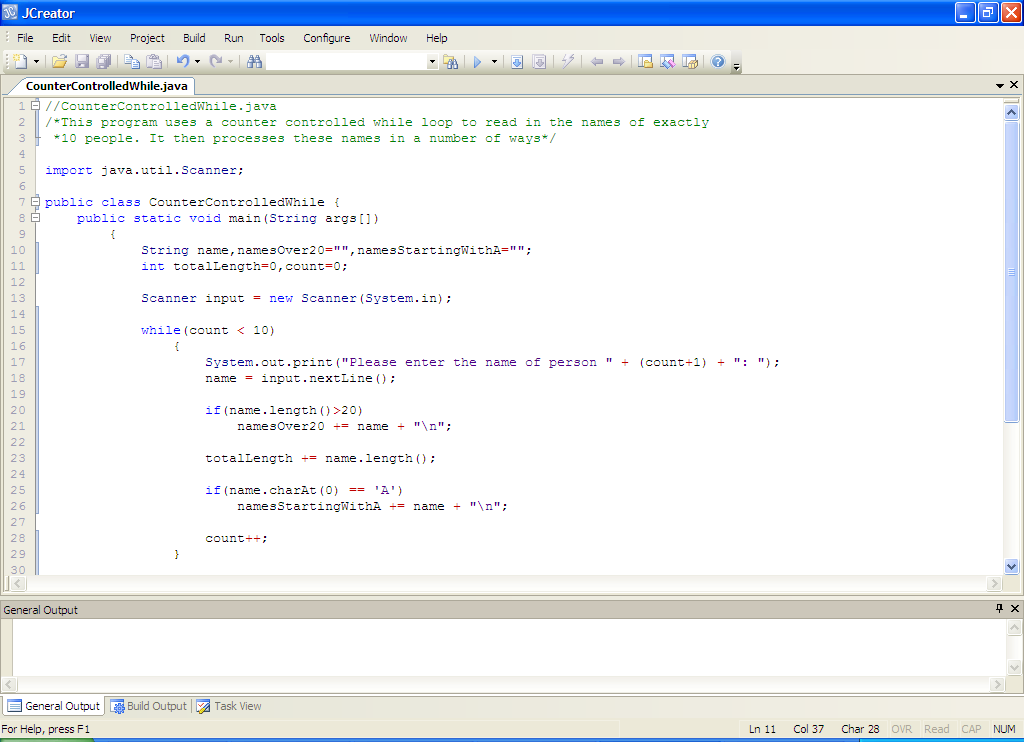
here *initial value* will **normally** be zero or one, *final value* **normally** corresponds to the **number of iterations** required, *statements1....n* refer to all the statements within the body of the while loop, and the statement which increases (increments) the value of *counter variable* also appears within the loop body, often towards the very end of it. Note that the increment value can also be negative here.

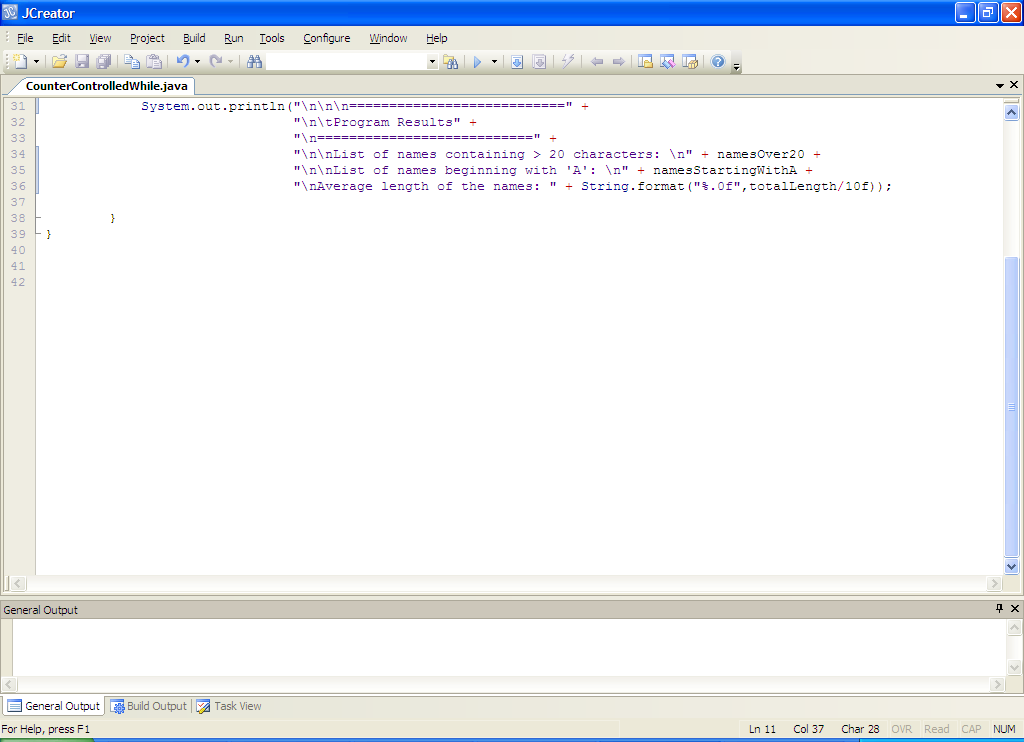
**Aim:** To write a program which uses a while loop for counter-controlled repetition to read in the names of exactly 10 people. When the loop completes the program will then display the list of names containing more than 20 characters, the list of names that begin with the letter “A” and the average number of characters in the names entered correct to the nearest whole number.

A sample run of the program produces the following:



**Java Code:**





**Analysis of program:**

• The program begins with a number of variable declarations/initialisations. Naturally there is a String variable called name to store each name as it is entered. There are also 2 other String variables here, namesOver20 and namesStartingWithA. The first of these variables is needed as the program is meant to **keep track of the names of anyone whose name length exceeds 20 characters** and the second one is needed as the program is meant to **keep track of the names of anyone whose name begins with ‘A’**. These variables **must be initialized to the empty string** **“”** here in order to allow them to carry out their job within the loop later. Failure to initialize either would result in a **syntax error**.

• There are also 2 integer variables. The first of these, totalLength, is initialized to zero. You’ve seen this before for numeric variables – this is because totalLength is meant to **keep a running total of the lengths of all the names entered** so that, once the loop finishes, we can calculate the average name length. The second integer variable, count, is simply referred to as a **loop counter variable**. Its main purpose is to **keep track of the number of times the while loop has iterated**, to ensure that it will come to a stop once 10 iterations have occurred. It is **also used in the prompt** where the user is asked to enter the name with the code **(count+1)**.

• The next main action is the while loop itself. In this case the counter-controlled while loop is set up to iterate a total of 10 times. It achieves this as follows:

**int count = 0;**

**while (count<10)**

**{**

**//other loop code**

**count++;**

**}**

So count begins with the value zero upon initialization (note that it could just as easily have been set to one – many people prefer this initialization value and it is perfectly acceptable as long as the test condition in the loop then becomes (count<=10) or (count<11). All that matters ultimately is that 10 iterations are achieved). Then a test is performed at the top of the while loop which compares the value of count with 10. If it turns out that count is < 10 (which it is at this point), then the loop body will execute. So all the statements inside the while loop will now execute. The very last statement in the loop body is

**count++;**

which has the effect of incrementing the variable count, so that its value increases from 0 to 1. Now execution jumps right back up to the top of the while loop and the test is performed again. count is still < 10 at this point so the process repeats over and over until the value of count eventually reaches 10. At that stage the test (count<10) becomes false as (10<10) is false and so the loop immediately stops and the execution jumps to the println() following the loop.

• Each iteration of the while begins by asking the user to enter the name of a particular person. This is read in using nextLine().

• The next chunk of code involves a simple if test which checks to see if the length of the name entered exceeds 20 characters. In Java, the length of any string can be obtained through the **length()** method. So here we have **name.length()** to tell us the length of the string stored in the variable name. This is a very useful method we will use a lot throughout the rest of this module and into the OOP1 module. If this test turns out to be true then we have encountered a name exceeding 20 characters in length and so we want the program to keep track of this name – to basically add it to a list of names that exceed that length. The adding of the name to the list is achieved with the code

**namesOver20 += name + “\n”;**

This uses the **+=** operator to join the value stored in name to the current contents of the string namesOver20. It then appends a newline onto the end of it (this has the effect of putting each name in the list on a new line of its own).

• Next up is the code which will keep track of the total of all the lengths of the names entered. Again the += operator is used here for efficiency. We use **name.length()** again here to give us the length of the name concerned.

• The next piece of code is another simple if test. This one checks to see whether or not the name just entered begins with a capital ‘A’. We use the **charAt(0)** method here to extract the first character from the name. Then we compare the value extracted with ‘A’. If the test turns out to be true then the name gets added onto a list of names that begin with ‘A’, in the variable **namesStartingWithA**.

• The loop ends with the code

**count++;**

which, as mentioned earlier, increments the loop counter variable. Without this code, the **loop could never end** and we would effectively have an **infinite loop**, a **runtime error**.

• Once the loop finishes, the results of the program are displayed.

**Organising your Work**

You should have a folder under X: called JavaStuff created. This time, create a folder called **Lab7** within JavaStuff to save your work from this lab session.

**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **CounterControlledWhile.java** in your Lab7 folder. Now type in the code for the program above.

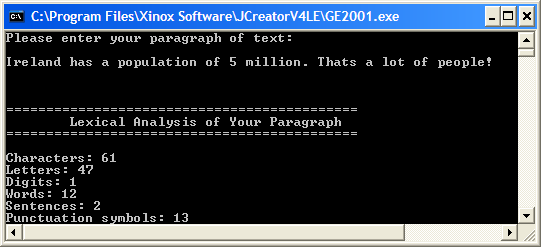
If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

Once you are free from errors, run the program and test it out fully.

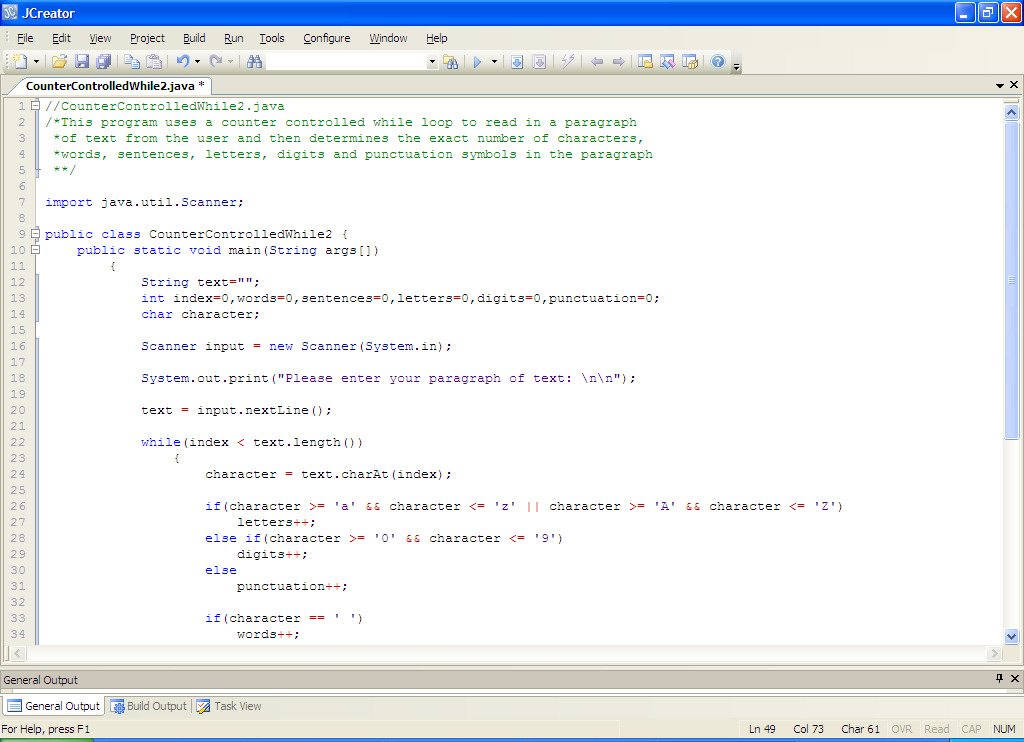
Now, you should **modify** the original program to momentarily **comment out the code that increments the variable count within the loop**. Now recompile and run the program. You should now appreciate the importance of this line of code. As mentioned recently, you are now dealing with a **runtime error**. Its great practice to tinker about with programs like this so that you get exposed to all sorts of errors. The notes you are studying here only give you the bare minimum really in terms of error types etc. You will learn so much more by modifying code yourself.

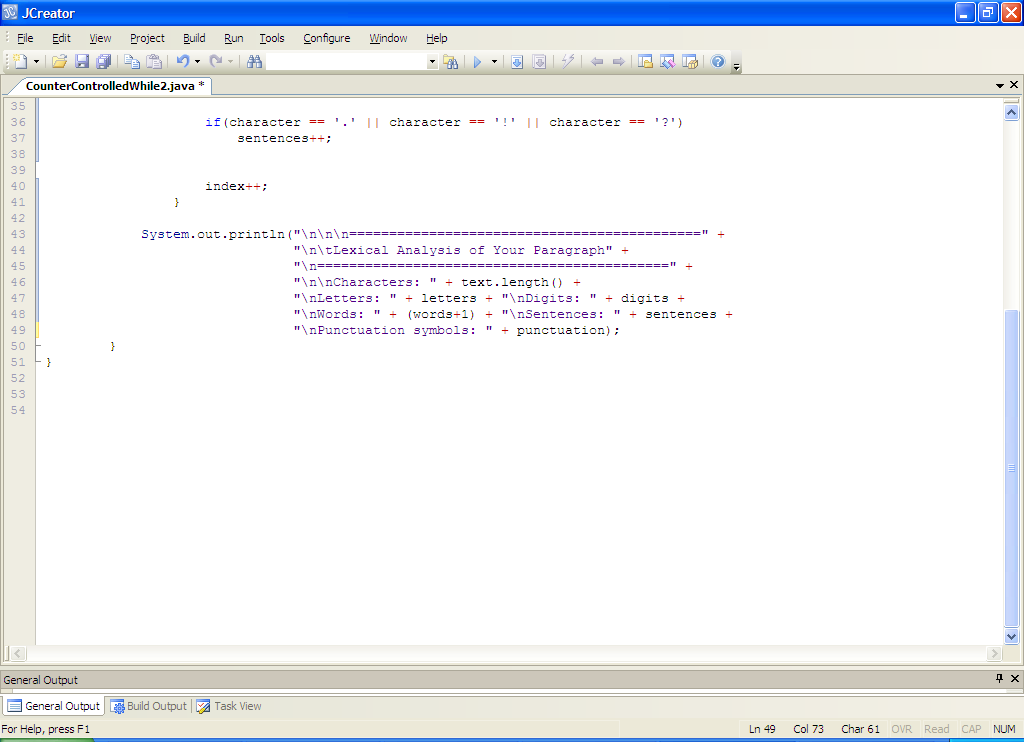
**Another Program Using a Counter-Controlled While Loop**

**Aim:** To write a program which uses a while loop for counter-controlled repetition to read in a paragraph of text from the user and determine the number of characters, letters, digits, punctuation symbols, words and sentences in the paragraph



**Java Code:**





**Analysis of program:**

• There are several variables used for keeping track of different quantities. All of these are initialized to zero. The variable **index** will be used as both a **loop counter** and as the **index value** used by the **charAt**() method to extract a character from the text entered by the user.

• The user is asked to enter their paragraph of text. The **number of characters contained in this text is found by using the length() method** as per the previous program we looked at.

• The next section of code contains the counter-controlled while loop. Note that the test condition is

**index < text.length()**

so here, the loop will continue to iterate as long as the value of index (which begins with the value 0) remains lower than the value of **text.length**(). So this loop will **iterate a set number of times**, determined by the amount of characters contained in the text entered by the user. The variable index is the **loop counter variable** in this case since its value changes as the loop progresses, through the code

**index++;**

and its value ultimately brings the loop to a halt (or at least it should!)

• The first thing to happen inside the loop is that a character is extracted from the text entered by the user through the code:

**character = text.charAt(index);**

The first time the loop iterates the value of index is zero so text.charAt(0) will extract the **first character** in the paragraph. At the end of the loop index is incremented, so the next time this code executes, text.charAt(1) will extract the second character in the paragraph. This process continues until all characters have been processed.

• With every iteration of the loop, a **nested else-if** structure executes. This determines the total number of letters, digits and punctuation symbols contained in the text entered. These are all **counter variables** in the program as they are used to keep a count of a certain quantity.

• Following this, there are 2 **simple if** structures. These are used to determine the number of words and sentences in the text. We take it here that **all words are separated by one blank space** and so the total number of words in the text will be equal to the total **number of blank spaces + 1** (we have to add one because the very last word won’t have the blank space following it so we must compensate for that). We take it that **all sentences must end with a ‘.’, ‘!’ or ‘?’** and the total number of these characters allows us to determine the number of sentences in the text. So there are bits of **problem-solving code** here to get your head around.

• As mentioned before, the code

**index++;**

allows the loop to eventually come to a halt - when the value of index eventually reaches the value of text.length() - but also allows the value of the variable character to be updated with each loop iteration as required for the program to function correctly.

• When the while loop has completed, the program then just displays the number of words, sentences, characters etc. to the screen

**Typing in and Saving Code for the Program Just Analysed**

Click the **New File** icon on the JCreator window, save the program as **CounterControlledWhile2.java** and away you go.

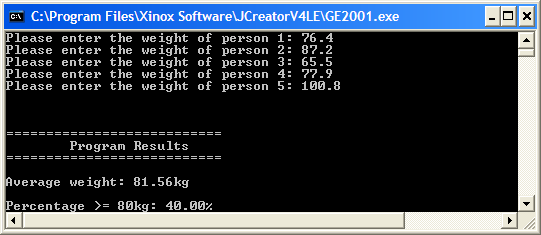
If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**). If you spot any differences correct them and compile again until the program is syntax error-free.

Once you are free from errors the program can be executed.

Now **test the program fully** by entering a few different lines of text and seeing do the figures tally.

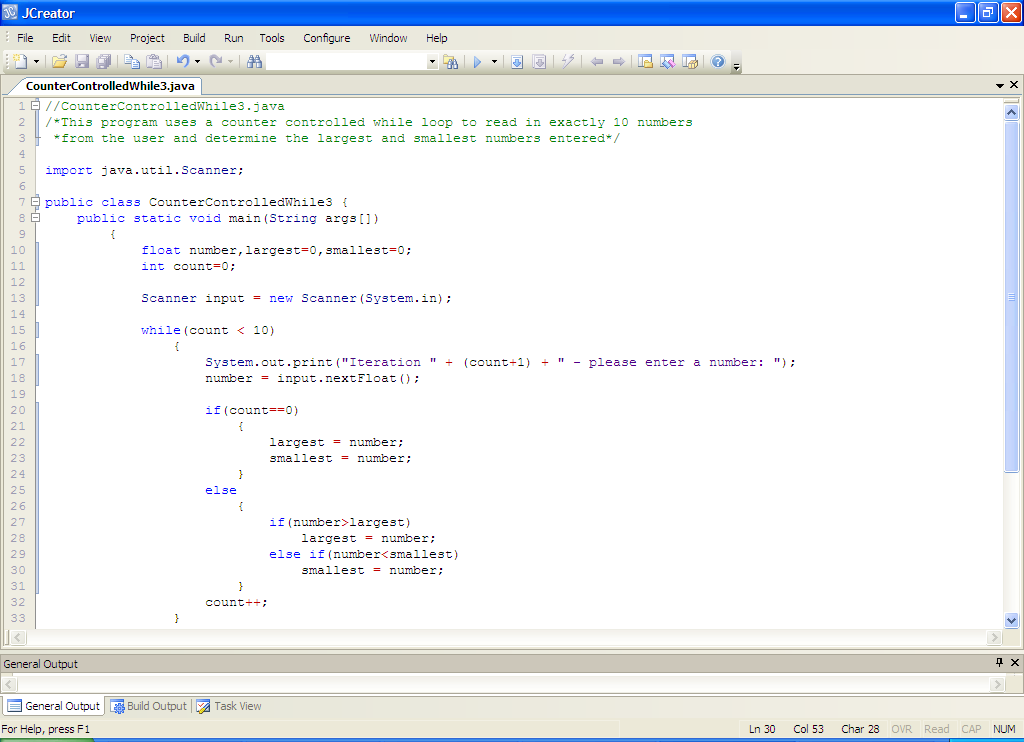
**Exercise 1**

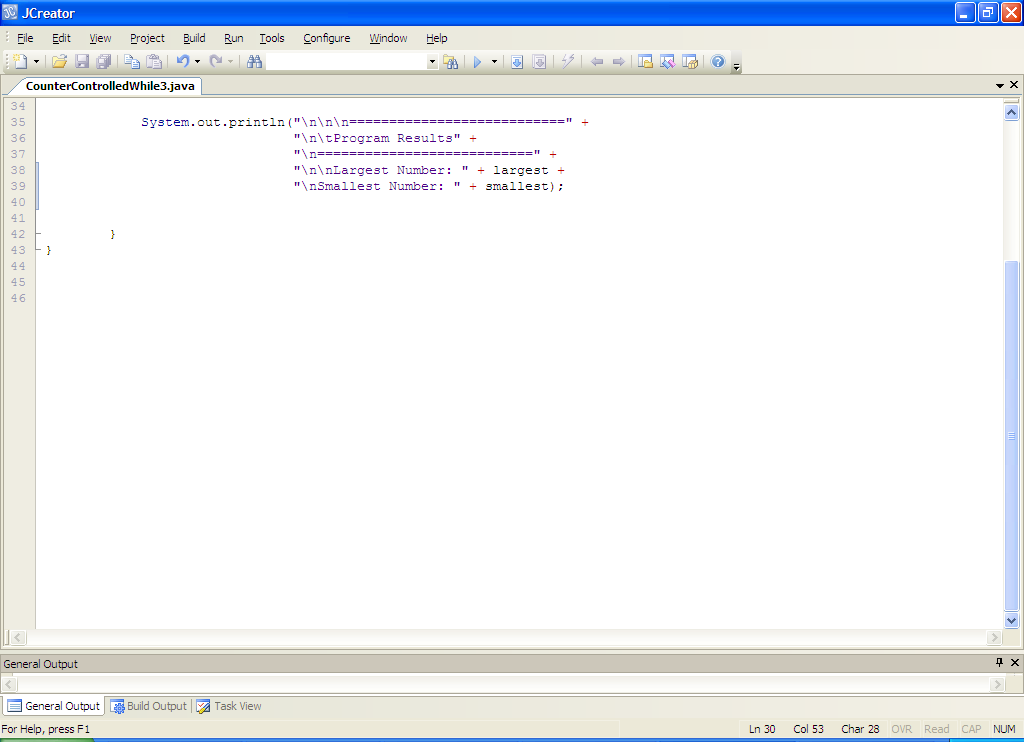
Write a program that uses a **counter-controlled while loop** to read in the weights of exactly 5 people. The program should determine the average weight along with the percentage of people whose weight is at least 80kg. It should display the average to **2 decimal places**. You should save your program as **Exercise1.java** and it should run according to the following sample screenshot:



**Another Counter-Controlled while Example**

**Aim:** The program reads in 10 user-supplied numbers and determines the largest and the smallest of the values entered.





**Analysis of program:**

• Within the loop, the user is asked to enter each number in turn and then an if-else structure forms the decision-making part of the program. The if section of this is:

**if (count == 0)**

**{**

**largest = number;**

**smallest = number;**

**}**

which checks to see if this is the very first iteration of the loop. If it is (it will be the first time the loop iterates as the value of count will be zero), then we wish to set the value of the variables largest and smallest to the very first value entered by the user because at this stage the first value entered must be the largest and the smallest simultaneously. Of course, this if test evaluates to false for all subsequent iterations because the value of count will be non-zero.

The else section deals with every other iteration after the first one. In this case the code

**else**

**{**

**if (number > largest)**

**largest = number;**

**else if (number < smallest)**

**smallest = number;**

**}**

contains an embedded if-else statement. The if section of this checks to see whether the current value of the variable number exceeds the current largest value entered. If it does then the variable largest is reset to the value of this number. The second works in a similar manner except it tracks the smallest value entered. With this code we always have the current largest and smallest values entered tracked as the loop iterates. Take some time to try to get your head around how this algorithm is working.

• The code

**count++;**

just increments the loop counter as usual.

• The program then displays the largest and smallest values entered.

**Typing in and Saving Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **CounterControlledWhile3.java** in your Lab7 folder. Now type in the code for the program above.

If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

Once you are free from errors, run the program several times with different sets of inputs.

**Note**: It can be tedious testing out a program like this that requires 10 input values to be supplied – it is a **very good idea to lower the number of inputs** to something more reasonable, such as 4 or 5 values. You can usually rest assured that if it works for 4, it will work equally well for 40000.

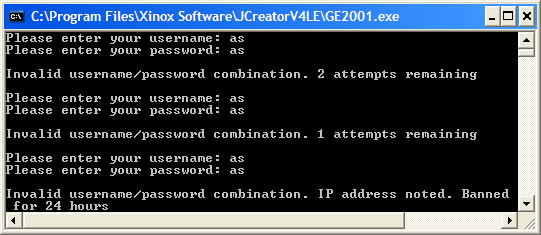
**Exercise 2**

In many computer systems, the user needs to supply a username/password combination to gain access. In many such systems, there is often a limit on the number of times a user is allowed to enter the username/password before they are banned from making further attempts to gain access for security reasons.

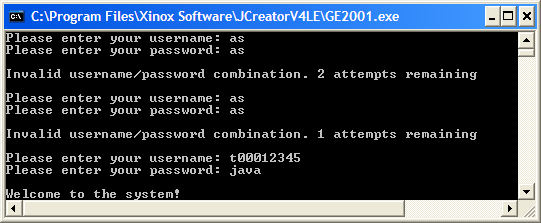
Write a program called **Exercise2.java** that asks the user for their username and password combination. Assume here that the correct username is “t00012345” and that the correct password is “java”. The user should be given **at most 3 attempts** to enter the correct username and password. If they fail to give the correct username/password combination the first 2 times an “Invalid username/password” message should appear and they should be told how many attempts they have left. If they fail the third time they should get a “banned” message as indicated below. If they enter the correct username/password they should receive a welcome message and the **loop should immediately terminate** (**hint:** this can be easily achieved by inserting a **break** statement in the correct location).

Your program should run according to the sample screenshots below:

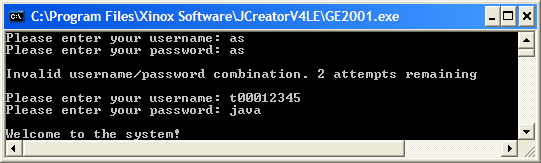
Run 1 – wrong on all attempts



Run 2 –right on the last attempt

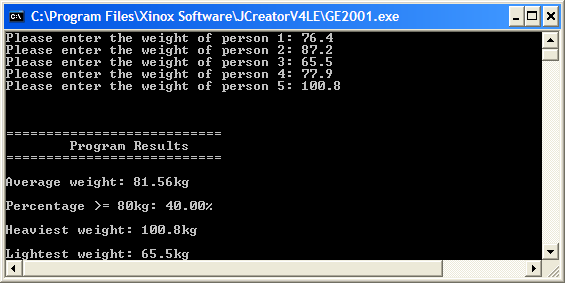


Run 3 – right on the second attempt



**Exercise 3**

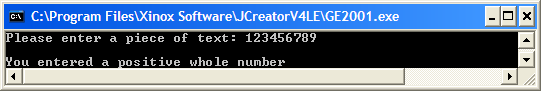
Take the **Exercise1.java** program you wrote earlier and save it as **Exercise3.java**. You should now **modify** the program so that it is capable of keeping track of the heaviest and lightest weights entered. It should then display this additional information as indicated in the screenshot below:



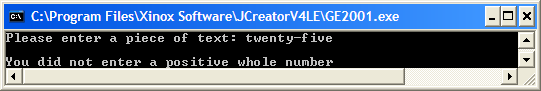
**Exercise 4**

Write a program that first of all reads in a piece of text from the user. The program should then analyse the text, character by character, to determine whether the character is a digit (use a **counter-controlled while loop** for this process). If a particular character is a digit, then the next character should be examined. Otherwise the loop should be immediately exited. When the loop has finished, a test should be carried out to compare the number of times the loop iterated with the length of the string entered. If it turns out that these values match, then the program should display the message “You entered a positive whole number”. Otherwise the program should display the message “You did not enter a positive whole number”. Your program should run as indicated in the following sample screenshots:

Run 1:

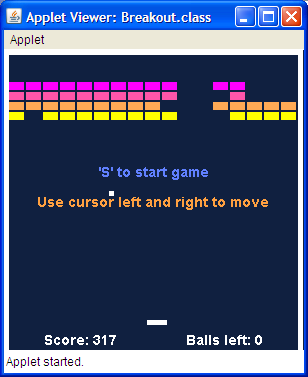


Run 2:



**Java in the Real World**

You have a lot of programming behind you at this stage so it’s a good time to have a look at another example of Java being used in the real world. The last time we saw Java being used as an educational aid, this time we’re going for a little bit more fun than complex numbers! Sit back for a few minutes and relax with a few games of Java **Breakout**, one of the arcade games of my childhood ☺



The version of the game we are going to look at was created by Brian Postma, whose Java stuff is at <http://www.brianpostma.com/java.html> Here you will find links to all sorts of games and other stuff written in Java. The link for the Breakout game is on the right of the page, about half way down. Click the “**Play the Game**” link to have a few games now.

Once you have had you fill of Breakout, it’s down to the serious stuff of examining the code that Brian used to create this game. I have downloaded this code already and it is in a folder called **Breakout** in the Lab7 folder.

Once you open the file **Breakout.java**, you’ll see that it contains a lot of code. In fact, Brian is unusual in that he has put all of the code for his game into just one Java file. You will discover in year 2/3 that it is good practice to split up an application like this into a number of separate Java files, as it makes for **easier code maintenance** in the long run.

Don’t be put off one bit by the code you see as an awful lot of it involves applet specific code and setting up the graphics end of the things which we never touch in first year - my goal here is to expose you to a real-world Java application and hopefully you will recognize various bits and pieces of the code as you look through it, such as the class definition, the variables, the fact that 2 packages are being imported, the use of constants with the final keyword etc.

Within the **paint**() method (starting on line 127) you see the following **if-else** statement:

**if (ingame)**

**PlayGame();**

**else**

**ShowIntroScreen();**

this is testing to see whether we are currently in a game. If we are, then the method PlayGame() is called, otherwise the method ShowIntroScreen() is called. Again, don’t worry about the details of these methods at all, the main thing is that you can recognize the if-else statement and understand how it operates.

The method **CheckBatBounce**() is interesting for us as it uses a number of structures that we have covered including an if-else, a while loop and a switch, as well as relational, logical, arithmetic, increment and arithmetic assignment operators. There are even calculations taking place within the method.

Spend a few minutes having a look at the code to see what else you can recognize.