**Institute of Technology Tralee**

**Computing Department**

**Introduction to Programming**

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

**Practical 6 – Looping with while**

There are just 3 categories of control structure in programming, the **sequence control structure** where statements just execute in sequence, one after another, the **selection control structures** like the **if-else** and the **switch** statements which allow different sets of instructions to be executed depending on whether certain conditions are satisfied or not and, finally, there are the **repetition control structures** which allow certain groups of instructions to repeat over and over again. We shall cover Java’s primary repetition control structure in this lab sheet – the **while loop**.

Take your time going through this lab sheet and **make sure you understand** what is going on – it is generally **at this point each year that students start finding things more difficult in programming, especially from a problem-solving perspective**. Making sure you can understand the algorithms involved in each program as you encounter them will help you to tackle future lab sheets and exercises much more easily.

**Why Loops?**

Loops are essential in structured programming. We have got away without them up until now but, in reality, software systems require them in general. Imagine that you needed to write a program to read in the height details of 8 students and process them in some way, for example, you wanted to determine the average height of the group of 8 students. In order to write this program you might come up with the following **pseudocode design**:

*Set height total to zero*

*Read in the height of the first student*

*Add this height value to the height total*

*Read in the height of the second student*

*Add this height value to the height total*

*Read in the height of the third student*

*Add this height value to the height total*

*Read in the height of the fourth student*

*Add this height value to the height total*

*Read in the height of the fifth student*

*Add this height value to the height total*

*Read in the height of the sixth student*

*Add this height value to the height total*

*Read in the height of the seventh student*

*Add this height value to the height total*

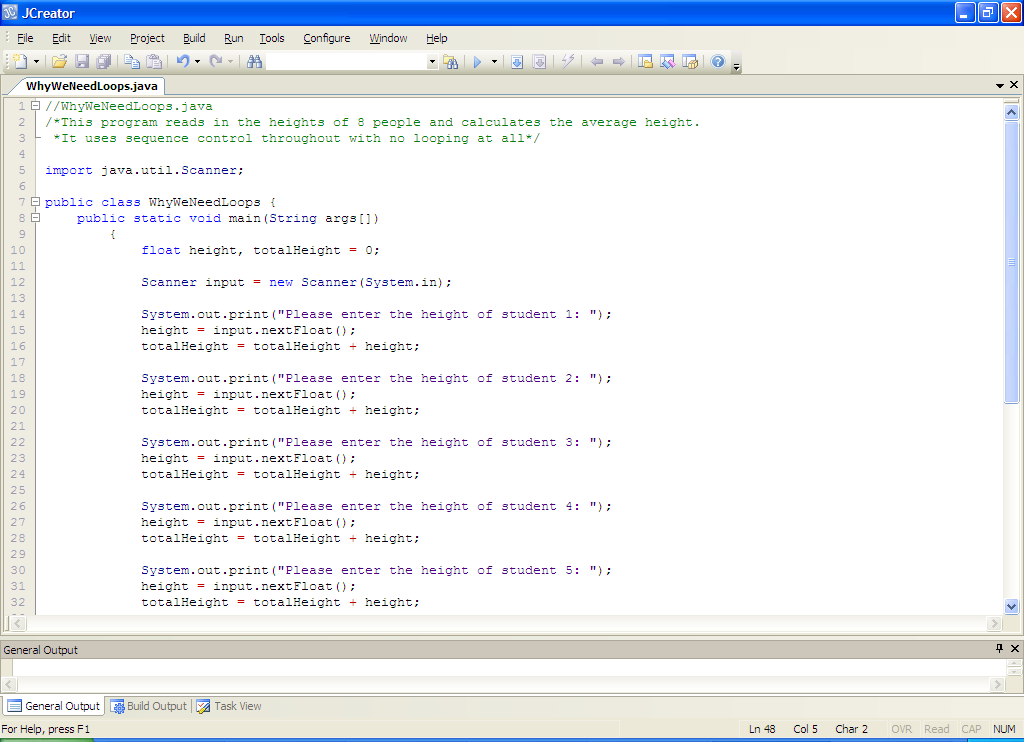
*Read in the height of the eight student*

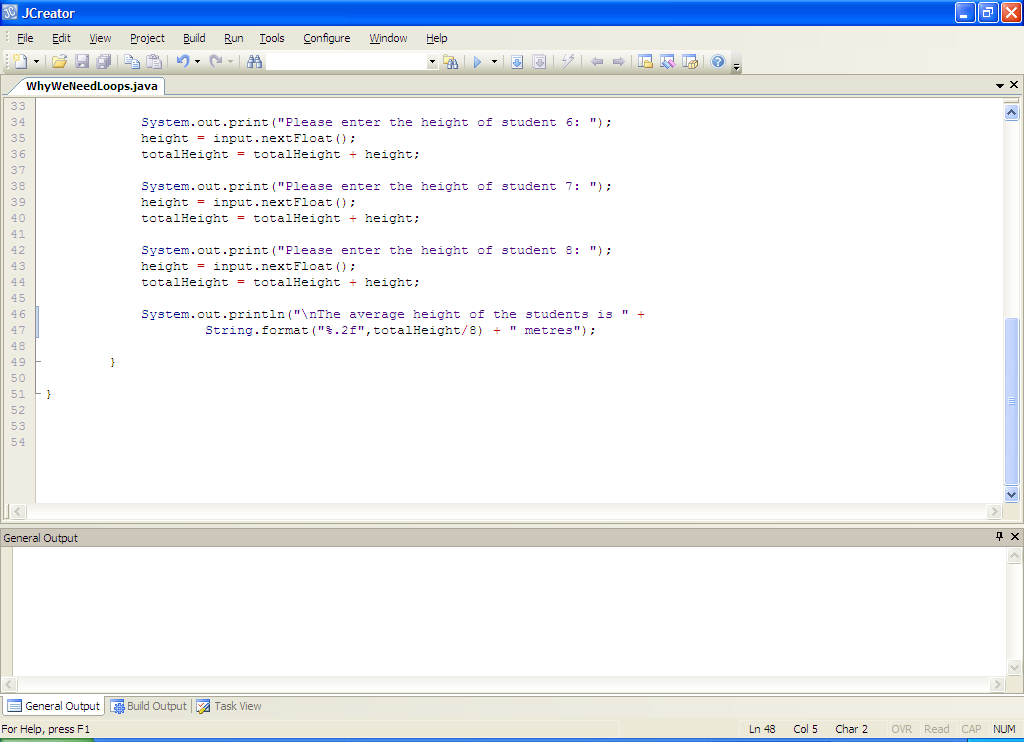
*Add this height value to the height total*

*Calculate the average height as height total divided by 8*

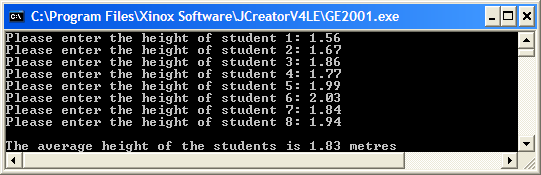
*Display the average height*

You might then go on to translate this pseudocode into actual Java code as follows (you don’t need to type out the following program unless you wish to for practice)





This program works perfectly well and it produces the correct output as indicated in the following sample run:



The program is called **WhyWeNeedLoops.java** and you’ll find it in the **Lab6 folder –** you should examine the code, compile and run it for yourself now.

Now, imagine that, instead of a group of 8 students, it is a group of 1000 students! Suddenly the amount of code for the program has ballooned out of all proportion. But at least it is still doable. But what if you don’t know in advance of writing the program how many students there will be? In order to overcome these issues, programmers need to be able to use **loops**.

The **main function** of loops is to **allow programmers to develop applications that are capable of processing large amounts of data using the minimum amount of code**. It is perfect for the scenario just described where the same repetitive tasks are being performed over and over again.

Loops can also be used, as we shall see, to **control an application’s interaction with the user**, repeatedly executing the same process until instructed by the user not to.

**The while Loop**

First of all we shall examine the while loop, which is Java’s **general-purpose** repetition control structure and **may be applied in several different ways** depending on the program requirements.

The **general form** of the while loop is as follows:

**while (***expression***)**

**{**

*statement1*;

:

*statementn*;

**}**

*expression*above refers to any valid Java expression and *statement1...statementn* refer to all the statements which are part of the looping structure. while is a **keyword** and the **parentheses** around the expression are essential. The **curly braces** are required whenever there are at least 2 statements in the loop.

The way in which the structure actually operates is as follows:

The while loop evaluates the expression. If it evaluates to true then all the statements that are part of its looping structure are executed. Following this first iteration, the expression is re-evaluated and if it continues to remain true the statements are re-executed. This process continues until the expression becomes false, at which time the loop stops and program execution continues at the statement immediately following the while loop.

**Three Applications of a while Loop**

The while loop is **general-purpose** and may be used in all cases where repetition is required in a Java program. There are 3 possible ‘types’ of while loop as follows:

• **data-sentinel controlled** while loop

• **counter-controlled** while loop

• **task-controlled** while loop

Each of these applies the while loop structure in a slightly different way so we will look at all 3 possibilities in some detail now, starting with the **data-sentinel** case. It is well worth spending time now trying to understand the operation of these loops because they are constantly used by programmers in their applications and are similar in all programming languages.

**Data-Sentinel Controlled while Loop**

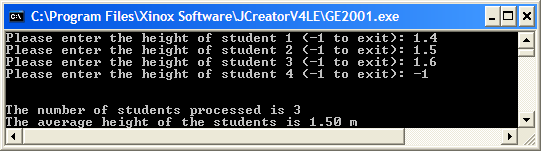
You may or may not have noticed that up until now all our programs have just dealt with processing **one set of data**. We always knew in advance how many values would be processed and we accounted for this as we wrote our program by creating a sufficient number of variables to hold information. This had the effect of making our programs **static** in the sense that the user of the program could not really control how many values they wished to enter (they were always told exactly what to enter). To make programs more **dynamic**, and **to give a greater level of control to the end-user**, a data-sentinel controlled while loop can be used.

We will make use of this type of loop when we are dealing with an **unpredictable, unknown** or **arbitrary** number of items.

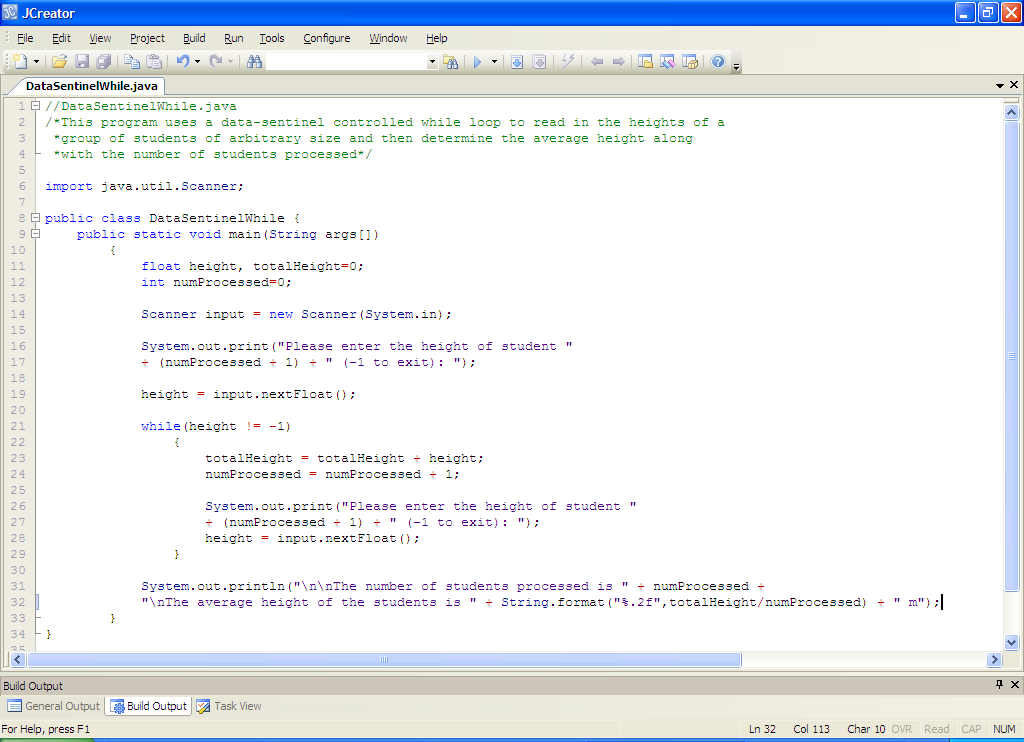
The **ATM machine** at the bank works on this principle, allowing you to begin a transaction, process your transaction and end your transaction before asking you if you would like to carry out further transactions i.e. a **dynamic** situation where the end-user is offered **greater flexibility** through the use of a repetition control structure. It is the end-user who decides whether or not they want the program to continue and not the program itself => **more user-friendly**.

**A First Data-Sentinel Controlled While Loop Example**

**Aim:** The purpose of this program is to use a data-sentinel controlled while loop to repetitively process the user’s input, which will be the heights of an arbitrary number of students. The program should display both the total number of students processed and the average height of the students processed to 2 decimal places. The program would run as indicated in the following sample screenshot:



**Java code:**



**Analysis of program:**

• The variables for the program are declared first of all and two of these are initialized to zero.

• After setting up the keyboard scanner, the user is asked to enter the height of the first student processed. When the program runs, the first output the user will see is the message

**“Please enter the height of student 1 (-1 to exit): ”**

Here height is referred to as the “**sentinel variable**” and -1 is the corresponding “**sentinel value**”. A sentinel is another word for a guardian and so it is that the variable height is guarding the loop in this case. Entering the value of -1 (the sentinel value) will being the loop to a halt.

Note the inclusion of the “(-1 to exit)” – this is **vital information** so that the user of the program actually knows what value needs to be entered so that the loop can be stopped. Without this, the user could spend a lot of time trying to guess the sentinel value!

-1 is chosen as the sentinel value here because all heights entered are expected to be positive numbers and so there will be **no conflict** of interest e.g. if I chose 1.5 as the sentinel value, the height of a student could also be 1.5 and so if you were to enter a height value of 1.5 the while loop would immediately stop, which is not what you want in that case. So it’s **important to choose your sentinel value carefully**.

Note that the number 1 appears after the word “student” as a result of the code

**numProcessed + 1**

Recall that numProcessed has previously been initialized to zero. Therefore to get out the “1” is just a case of adding 1 to the value of numProcessed. Note that when this code has executed, the value stored in the variable numProcessed is still 0. You might be inclined to think that this code would have the effect of increasing the value of the variable by 1 but it doesn’t. All that happens here is that the value of numProcessed + 1 is getting displayed. If you really wanted to add 1 to the value of the variable numProcessed you would have to use the **assignment operator** = e.g.

**numProcessed = numProcessed + 1;**

as we see later on in the program

• The next section of the program contains the while loop. This loop begins by performing a test. In this case the test compares the value of variable height to -1. If the first value of height entered by the user is actually -1 then the condition is true and the statements within the loop would never get executed at all. However, if the first height value entered is not -1 then the statements within the loop do get executed.

The first statement in the loop

**totalHeight = totalHeight + height;**

might look a little strange at first since it has totalHeight on both the left and right-hand sides of the assignment operator. This statement is used to **keep a total of all the height values that have been entered to date**. As in the case of numProcessed, the variable totalHeight is initialized to zero. When this statement executes for the first time, it adds the first value of height entered by the user to the value of totalHeight (currently 0) and then stores the result of this in totalHeight. So, when this statement executes for the first time, the value of totalHeight will be equal to the first value entered for height.

Of course, the reason **we need to keep a total of all the heights entered is because the program is expected to calculate the average height** of all the students processed. This average can only be determined if we have this total.

• The next statement in the loop

**numProcessed = numProcessed + 1;**

adds 1 to the current value of the variable numProcessed, which for the first iteration of the loop, is 0. The result of this addition (1) is then stored in numProcessed. This has the effect of incrementing the value of numProcessed. The next time the loop iterates, the value of numProcessed will now be 1, 1 will again be added to numProcessed (now 1 itself) to give 2 and this process continues until the loop stops.

This statement is **needed by us here to allow us to keep track of the total number of students processed**.

• The final 2 statements in the loop:

**System.out.print("Please enter the height of student "**

**+ (numProcessed + 1) + " (-1 to exit): ");**

**height = input.nextFloat();**

repeat the statements that immediately preceded the loop. It asks the user to enter the height of another student. This line of code is **essential** in the loop because it allows the loop to be stopped, whenever the user eventually enters -1 for the height. Without this statement, the user could enter the loop but could never exit it – giving an **infinite loop**, resulting in a **program crash** and **runtime error**.

• After the loop the println() displays the total number of students processed and the average height of the students to 2 decimal places via the format() method. Note that the **average height is calculated by dividing the total height by the total number of students processed**.

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

**N.B. Before you try to compile and run a program involving a loop make absolutely sure that you have your program saved first. Otherwise, if the program crashes, you may lose your unsaved program.**

**If your program does have an infinite loop, then you should try either closing the console (black) output window or clicking the lightning bolt icon**  **near the top of the JCreator Window.**

Create a new folder called **Lab6** within your **JavaStuff** folder to save your work from this week. Click the **New File** icon on the JCreator IDE and save the file as **DataSentinelWhile.java**. 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. Make sure that you **test it with several different sets of input values**. Use the **calculator** also to make sure the averages are correct. Also try getting the loop to stop by entering a value such as -2. Does this work? Try also to enter a piece of text such as “twenty” for one of the height values. What happens?

It’s important to see that, even though we have a program that functions at a certain level, it is certainly far from perfect. As we go on, we will cover the extra bits of code that we need to use in order to try to make our programs “fool-proof”. We will cover one of those pieces very shortly.

**General Form of Data-Sentinel Controlled while Loop**

In pseudocode terms, the data-sentinel controlled while loops have the following **general form**

*prompt for and read in the first item of data into the sentinel variable*

*while (sentinel value has not been entered for the sentinel variable)*

*process data in some way and carry out some other statements*

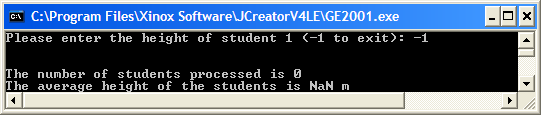
*prompt for and read in the next item of data into the sentinel variable*

Note that the **sentinel value** may be a single value, such as -1 in this particular program, but it **could also be one of a number of specific values or even a whole range of values**.

**Issue with the Last Program**

Some of you may have come across a **logical error** when you ran the last program. The problem occurs only if you enter no height values at all i.e. when prompted for the height of student 1, you enter -1 straight away. In this particular case, the loop statements never execute and therefore the value of the variable numProcessed remains at 0, which is of course correct since there were no students heights processed.

However, the println() method deals with displaying the average height and note that this is determined by dividing by the value of numProcessed, which is now 0. The result in this case is:



As the result being displayed here is a floating-point value, Java avoids a runtime error by displaying the result as **NaN** – short for **Not a Number**. **Division by zero is illegal** in all programming languages – even on your calculator! We can do better than this however

through good programming as follows:

if (numProcessed != 0)

System.out.println("\n\nThe number of students processed is " + numProcessed +

"\nThe average height of the students is " +

String.format("%.2f",totalHeight/numProcessed) + " m");

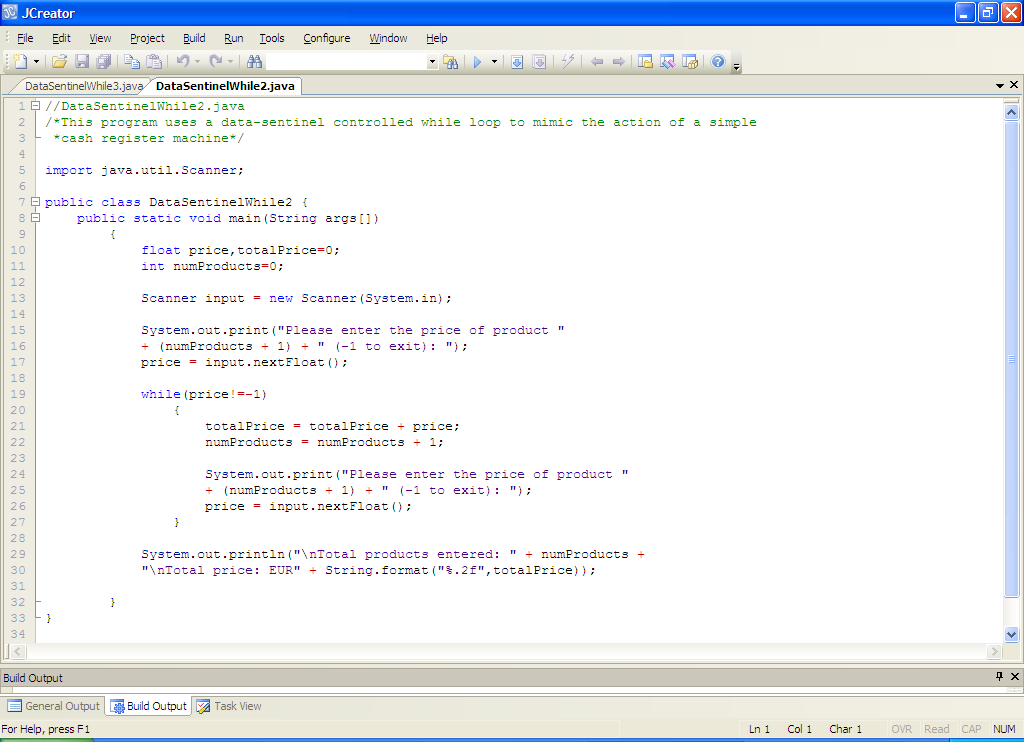
This code simply adds a **simple if** to check whether the value of numProcessed is 0 and, if so, it does not attempt to execute the println() call. Hopefully you see it makes good sense not to attempt to output the average because you have no average to output really.

You should now **modify** your existing DataSentinelWhile.java to include this extra test, compile and run the code again and try entering -1 on the first prompt.

**Another Program that uses a Data-Sentinel Controlled while loop**

**Aim:** The purpose of this program is to use a data-sentinel controlled while loop to repetitively process the user’s input, which will be the prices of an unknown number of products. In this way, the program mimics a simple cash register machine in a shop. The program should display both the total number of products processed and the total price of all the products.

**Java Code:**



**Analysis of program:**

• After the variable declarations/initialisations, the user is asked to enter the price of the first product.

Here price is the “**sentinel variable**” and -1 is the corresponding “**sentinel value**”. Entering -1 for the price will bring the loop to a halt here.

Again, note the inclusion of the “(-1 to exit)” – this is vital information so that the user of the program actually knows what value needs to be entered so that the loop can be stopped.

• The next section of the program contains the while loop. This loop begins by performing a test. In this case the test compares the value of price to -1. If the first value of height entered by the user is actually -1 then the condition is true and the statements within the loop will never get executed at all. However, if the first value entered is not -1 then the statements within the loop do get executed.

The first statement in the loop

**totalPrice = totalPrice + price;**

is used to **keep a total of all the price values that have been entered to date**. The variable totalPrice starts off with an initial value of 0. When this statement executes for the first time, it adds the first value of price entered by the user to the value of totalPrice (currently 0) and then stores the result of this in totalPrice. So, when this statement executes for the first time, the value of totalPrice will be equal to the first value entered for price.

The reason **we need to keep a total of all the prices entered is because the program is expected to display this quantity** in the end.

• The next statement in the loop

**numProducts = numProducts + 1;**

adds 1 to the current value of the variable numProducts, which, for the first iteration of the loop, is 0. The result of this addition (1) is then stored in numProducts. This has the effect of incrementing the value of numProducts. The next time the loop iterates, the value of numProducts will now be 1, 1 will again be added to numProducts (now 1 itself) to give 2 and this process continues until the loop stops.

This statement is **needed by us here to allow us to keep track of the total number of products processed**.

• The final part of the loop repeats the statements that immediately preceded the loop. It asks the user to enter the price of another product. This line of code is essential in the loop because it **allows the loop to be stopped**, whenever the user eventually enters -1 for the price. Without this statement, the user could enter the loop but could never exit it – giving an **infinite loop**, resulting in a **program crash** and **runtime error**.

• After the loop is a println() method call, which just displays the total number of products processed along with the total price of all the products to 2 decimal places via the String.format() method.

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

Click the **New File** icon on the JCreator IDE and save the file as **DataSentinelWhile2.java** in your Lab6 folder. Now, for practice, 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. Make sure that you **test** it with several different sets of input values.

**Arithmetic Assignment Operators**

You will have noticed that the program above involved keeping track of certain quantities. The code

**totalPrice = totalPrice + price;**

was used to keep a running total of all the prices entered. Because this type of totaling is such a common thing in programming, many languages have special operators to deal with it, and Java is one of those. The above code can be written more efficiently as follows:

**totalPrice += price;**

It operates in exactly the same manner but is more succinct and is what you should try to use from now on in your programs where appropriate.

The operator above, **+=**, is called the **arithmetic assignment operator** **for addition**. There is also one for the other 4 arithmetic operators as follows: -=, \*=, /= and %=

**Increment/Decrement Operators**

You will also have noticed the code

**numProducts = numProducts + 1;**

used to keep track of the number of products entered. Because adding 1 to a variable is such a common occurrence, especially in programs involving repetition, Java includes a special operator to deal with it. This is called the **increment operator** and using this, the above code can be written more efficiently as:

**numProducts++;**

It should be noted that there are 2 types of increment operator – one is called **post-increment** (the one above) and the other is **pre-increment** which is written as

**++numProducts;**

**With pre-increment the value of the variable is incremented before the line of code executes whereas with post-increment the value is only incremented after the line of code executes** e.g.

**int a = 5, b = 4,sum;**

**sum = a++ + b;**

**System.out.println(sum);**

In this case the value of sum gets displayed as 9 because the increase to a only happens after the line of code has executed.

**int a = 5, b = 4,sum;**

**sum = ++a + b;**

**System.out.println(sum);**

In this case the value of sum gets displayed as 10 because the increase to a happens before the line of code has executed.

**Note that in our case here, it actually doesn’t matter whether we use pre- or post-increment since there is no other variable or value involved.**

To go along with the increment operator, there is also a **decrement** operator. This is **--** and can be pre- or post-decrement e.g.

**int a = 5;**

**a--;**

**System.out.println(a);**

Display the value of a as 4.

**Improving the Previous Program**

You should now take the DataSentinelWhile2.java program and save it as **DataSentinelWhile2Improved.java**. Armed with the information in the last 2 sections you should modify the code to make it more efficient. It should still run exactly the same as before.

**Exercise 1**

Write a program called Exercise1.java that converts from degrees Fahrenheit to degrees Celcius. The Celcius temperatures should be displayed to **3 decimal places**. The program will contain a data-sentinel controlled while loop which repeats automatically for an arbitrary number of conversions, **terminating when a temperature of *less than* -459.67 is entered**. When the loop terminates, the program should display

* the total number of conversions that were performed
* the average of the Celcius temperatures
* the total number of Celcius temperatures that exceeded 30C
* the total number of Celcius temperatures that were below 10C
* the percentage of Celcius temperatures that were at least 20C (to the **nearest whole number**)

as long as at least one conversion has been carried out. Otherwise it should just issue the message “No conversions were performed …. Goodbye”.

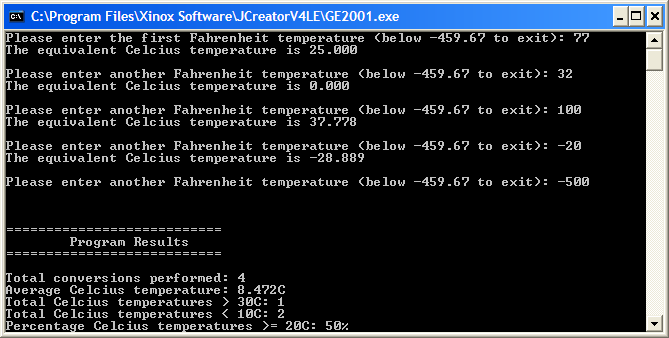
Try to use the **arithmetic assignment operator for addition** and **increment operator** where possible in your solution.

**N.B.** The formula for converting Fahrenheit to Celcius is as follows

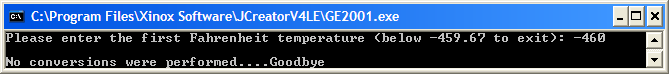


Your program should operate as indicated in the following sample screenshot. You **should use the values from the screenshot to test out your program fully**.

Run 1:

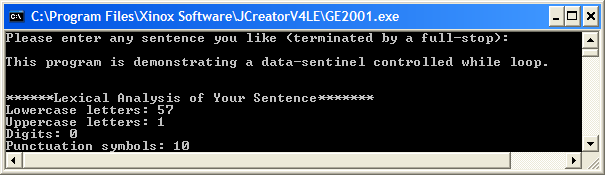


Run 2:

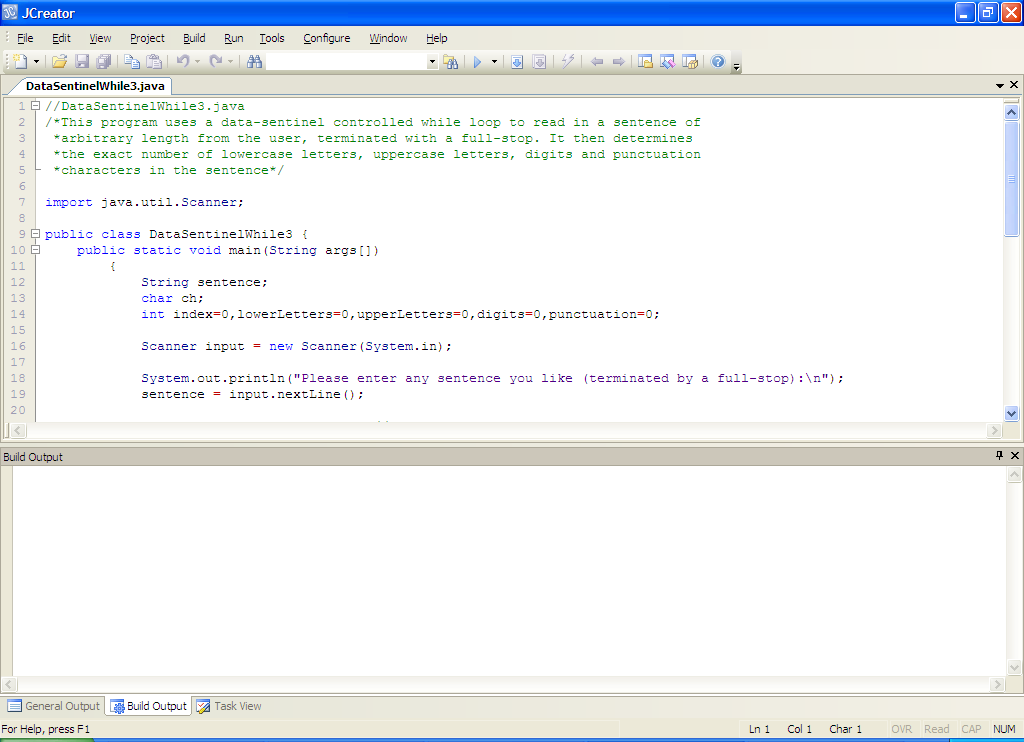


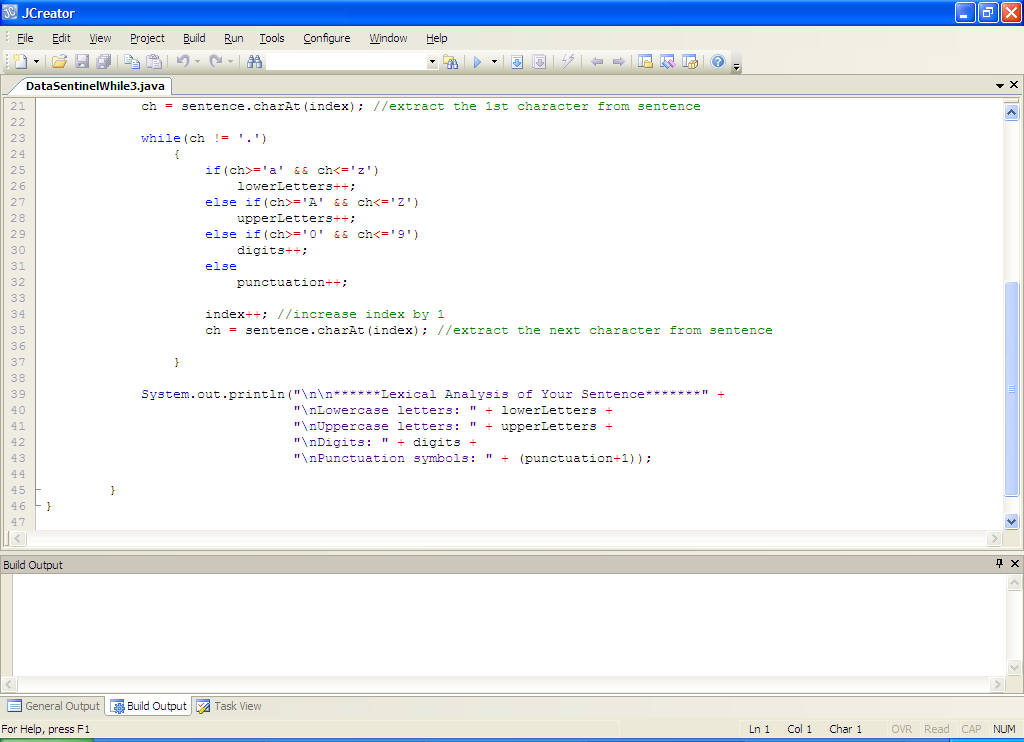
**Another Data-Sentinel Controlled while Loop Example**

**Aim:** The purpose of this program is to use a data-sentinel controlled while loop to repetitively process the user’s input, which will be a sentence of arbitrary length terminated by a full-stop. The program should determine the exact number of lowercase letters, uppercase letters, digits and punctuation characters contained in the sentence. It would run as indicated in the following sample screenshot:



**Java code:**





**Analysis of program:**

• A number of counter variables are declared to store the number of lowercase letters, uppercase letters, digits and punctuation symbols. These are all initialized to zero.

• The user is asked to input a sentence terminated by a full-stop, which is stored in the String variable sentence.

• The sentence is read in using **nextLine**() and then the first character of the sentence is extracted using the **charAt**() method. The value of the variable index at this point is zero so **charAt(0)**ensures the first character is extracted.

Similarly,

charAt(1) would extract the 2nd character in the sentence

charAt(5) would extract the 6th character in the sentence etc.

• The next section of code involves a while loop – this is the data processing loop in the program. You can see that it is data-sentinel controlled with **sentinel variable** being **ch** and **sentinel value** being **‘.’** The loop keeps going here until eventually ch stores the full-stop character.

Within the loop there is a nested else-if structure which keeps track of the number of lowercase, uppercase, digits and punctuation characters contained in the sentence. Note that all these variables are referred to as **counters** since that is their purpose. If a particular character in the sentence falls into a particular category, then its counter is incremented by 1. See the use of the **character comparison code** which is used in the test to determine the category of a particular character. Recall this uses **ASCII codes** of the characters as a basis for comparison. Note that **character constants must always be enclosed in single-quotes** e.g. ‘A’, ‘z’, ‘9’ etc.

• After the nested else-if structure is the code

**index++;**

**ch = sentence.charAt(index);**

the first line of code here simply increments the value of the index variable by 1 each time the loop executes. This ensures that in the next line of code, the charAt() method extracts the next character from the sentence and the value stored in ch is updated. Without this code, the **loop would never get a chance to exit** and we would be stuck with an **infinite loop** where the program would **crash**.

• When the loop has exited, the program displays the results. Note that 1 is added to the value of the punctuation variable to account for the full-stop character itself.

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

Type in the code for the program above (I know it’s tough but it’s all about practice – don’t forget your **copy and paste** facility too), save it as **DataSentinelWhile3.java** and then compile it, debug it and run it. **Test** it fully to make sure there are no logical errors or infinite loops.

**Exercise 2**

Take the DataSentinelWhile3.java program you have just dealt with and save it as **Exercise2.java**. Modify the original program so that the sentence can be terminated by a full-stop, an exclamation mark or a question mark. It should operate the same in every other way. A sample screenshot of the program would be:

