**Institute of Technology Tralee**

**Computing Department**

**Introduction to Programming**

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

**Practical 9 – Looping with for**

We’re almost there! You’ve now seen practically all of the structured programming aspects of Java in action and hopefully you’ve got a good handle on all the syntax we’ve covered so far as well as improved problem-solving skills – remember it’s all about practice, so keep coding, coding, coding! To finish off the lab course, today you will examine Java’s final repetition control structure, the **for** loop.

**The for Loop**

We have already encountered **counter-controlled repetition** as part of our study of while and do-while loops. There is, however, another means by which we can perform this type of repetition in a Java program. The **for loop** allows us to achieve counter-controlled repetition in a **more efficient** manner than with the other forms of loop – indeed it **exists specifically for counter-controlled iteration scenarios**. The general form of this looping structure is as follows:

**for** (*counter* ***=*** *startValue* **;** *counter* **<** *endValue* **;** *increment counter*)

**{**

*statement1*

:

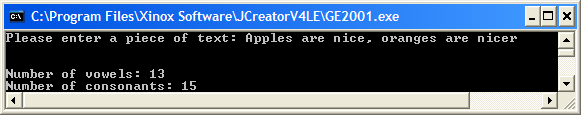
*statementn*

**}**

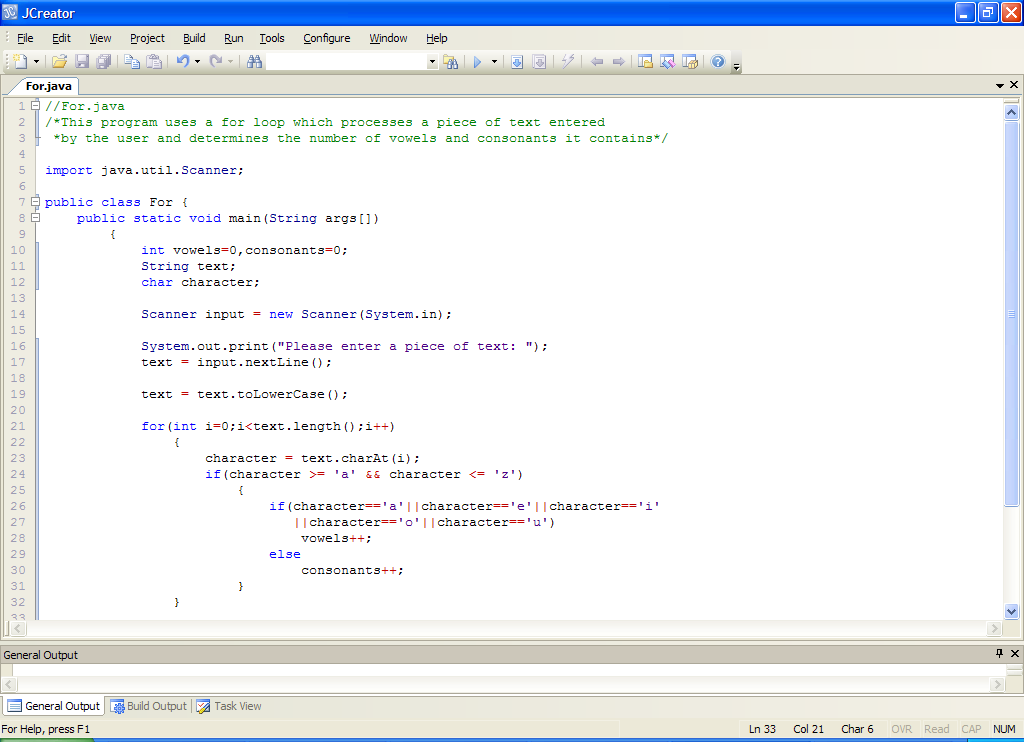
Here *counter* refers to the loop **counter variable**, which as always, keeps track of the number of loop iterations. *startValue* refers to the **initial value** of the counter variable – this is quite often 0 or 1. *endValue* refers to the **final value** of the counter variable which, when reached, causes the for loop to terminate. *increment counter* refers to the code that will increase (or decrease) the value of the counter variable each time the loop iterates.

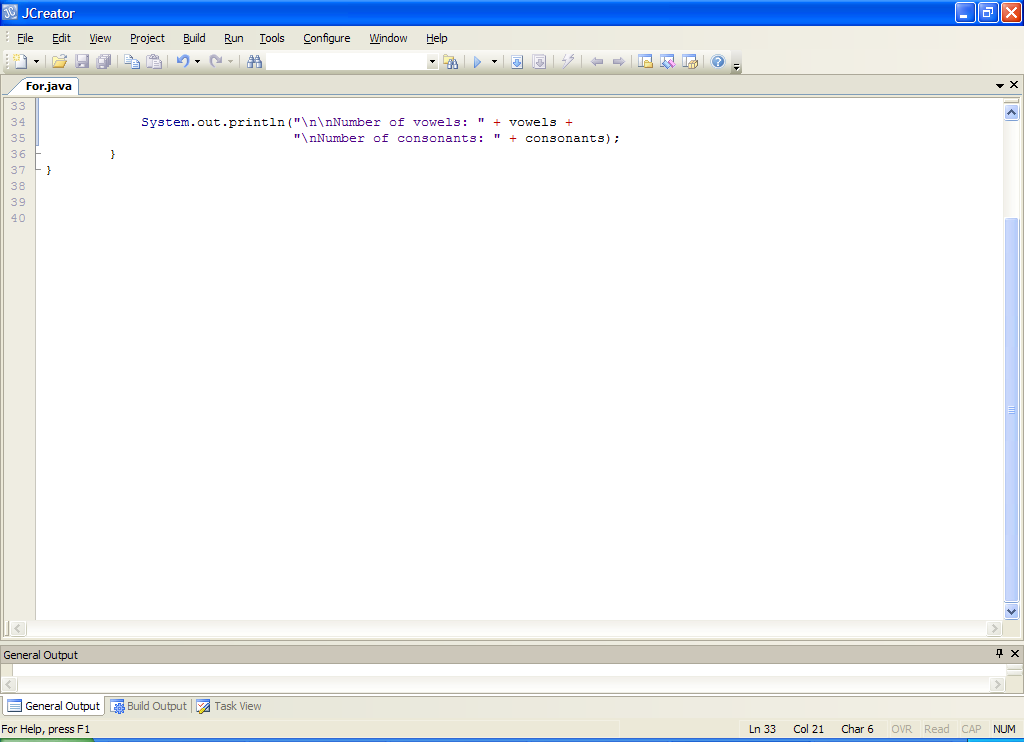
The word for is a **keyword** in the Java language and the parentheses and **semi-colons** are essential. The curly braces are needed if there is more than one statement associated with the loop.

**Aim:** To write a program using a **for** **loop** which will read in a piece of text from the user and determine the number of vowels and consonants it contains. The program would run as follows:



**Java Code:**





**Analysis of program:**

• Two counter variables are declared to store the number of vowels and consonants in the piece of text. These are initialized to zero.

• The user is prompted for the text which is then read in. It is then converted to lowercase form through the use of the **toLowerCase**() method. The reason for doing this may not seem obvious at this stage but the idea is to **make our test expression more efficient** later on when we are trying to determine if a particular character is a letter.

• Next we have the for loop. It has the following form here

**for(int i=0;i<text.length();i++)**

**{**

**//loop code**

**}**

The loop counter variable is **i** here. We’ve tended to use the name **count** to refer to loop counter variables until now but there is a bit of a tradition in programming circles to use the name i instead. You could, of course, have used count again here if you wanted to. Although i has no intrinsic meaning as such, it is in such common use that programmers automatically recognize it as a loop counter variable when they see it in code.

Here i is initialized to zero at the top of the loop. Note that this **initialisation will only happen once**. The value of i is then compared with the value of **text.length()** in the loop test condition. text.length() gives the number of characters contained in the piece of text just entered by the user. As long as i remains less than this value, the for loop will keep going and it will then process all the statements within the loop body, one by one. Once the very last statement in the loop body has executed, then i will increase by 1 with the code **i++**. In this way, the loop counter value will eventually get big enough so that the test condition becomes false and the loop stops.

• Each time the for loop iterates, the first thing that happens is that a character is extracted from the text supplied by the user. As you know by now, the method **charAt**() is used to do the extraction. The value extracted is stored in character.

• Next up we have a simple **if** test which determines whether the character in question is a lowercase letter. If it is, then we test further using an **if-else** to determine whether this letter is a vowel or a consonant. Depending on the outcome of this test, either the vowels or consonants variable gets incremented.

• Once the for loop completes, the program then displays the number of vowels and consonants found.

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

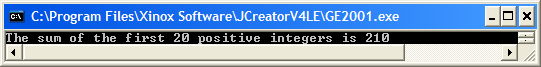
Create a new folder for this weeks work called **Lab9** within your **JavaStuff** folder. Click the **New File** icon on the JCreator IDE and save the file as **For.java** in your Lab9 folder. Now type in the code for the program above.

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 fully** for a number of pieces of input text.

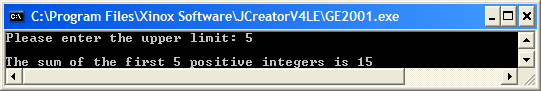
**Exercise 1**

Write a program called Exercise1.java that uses a **for loop** to calculate the sum of the first 20 positive integers, and then display this sum. Your program should run as follows – note that there is **no user input** in this program.



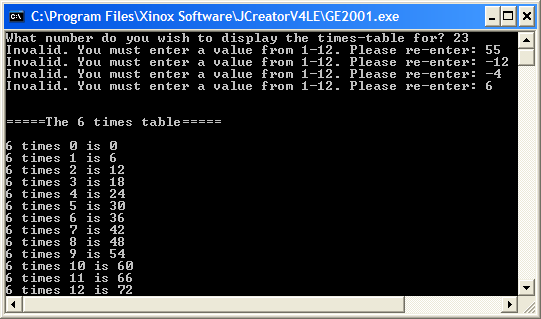
**Exercise 2**

Write a program called Exercise2.java that modifies the last program so that the user is allowed to enter a positive integer number and the program then calculates and displays the sum of the positive integers up to and including that upper limit. The program will run as indicated in the following sample screenshot:



**Exercise 3**

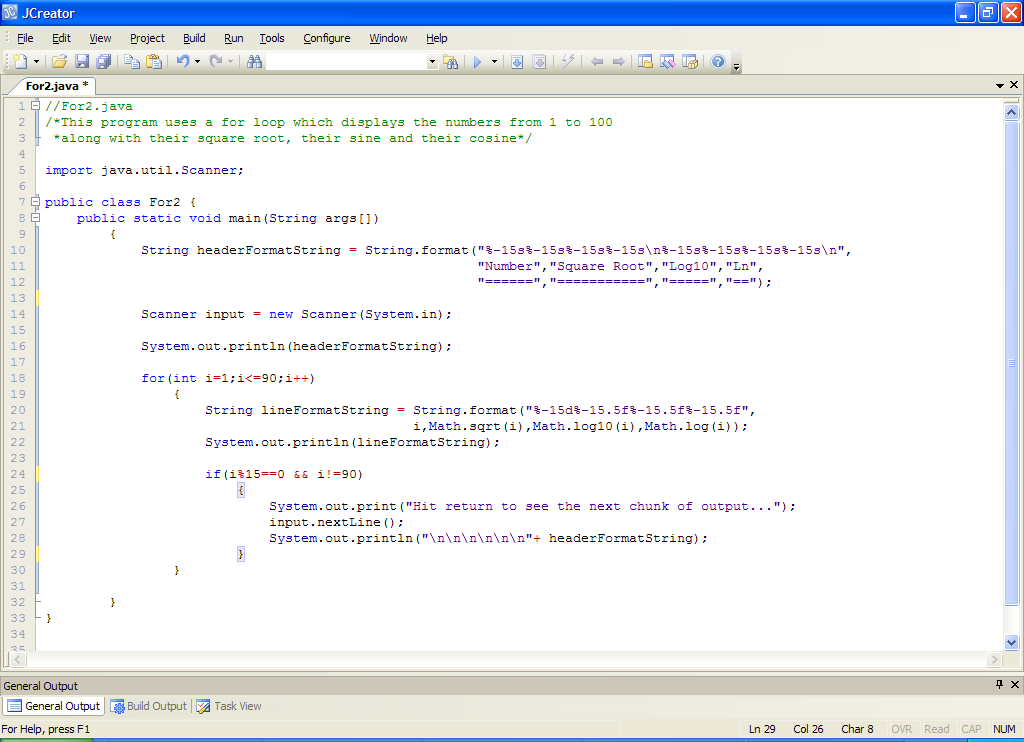
In primary school, children are taught their times-table to learn basic multiplication. Write a Java program called Exercise3.java that uses a **for loop** to display the n times table where n can be any whole number between 1 and 12 inclusive. The program should have some basic **validation code** here so that if the user does not enter a value for n within the range 1-12, they are given a warning message and continually asked to re-enter (you can use a **while** loop for this validation part). Once a valid value is supplied, the for loop then displays the table, line by line as indicated in the sample screenshot below:



**Another Program using a for loop**

**Aim:** We wish to write a program using a for loop which produces as output a neat table containing a number, its square root, its log to base 10 and its natural log for all of the numbers from 1 to 90, in chunks of 15.

**Java Code:**



**Analysis of program:**

• This program involves trying to set up a nice neat table of values. We take advantage of the **String.format**() method here in order to do the job. The variable **headerFormatString** holds the format string for the table header. Each column is given exactly 15 characters to display, in a left-justified manner. Recall the format specifier ‘**s**’ refers to the fact that the information to be formatted are all strings in this case, 8 of them in total.

• Before the loop begins, the table header is displayed – this is because we only want the header to come out once for a given chunk of output, not every time the loop iterates.

• the **for** loop comes next and it is set up to go around exactly 90 times, through the counter variable i.

• The first action inside the loop involves setting up a format string for each line of the output. This must correspond to the format string created previously for the table header so that the output appears neat and aligned nicely. Recall the format specifier ‘**d**’ for integer formatting (the number in this case) and the rest all use the format specifier ‘**f**’ since they are all **double** values – all the **Math** class methods in Java return double results. All the double values are displayed here to **5 decimal places** through the **.5** before the ‘f’.

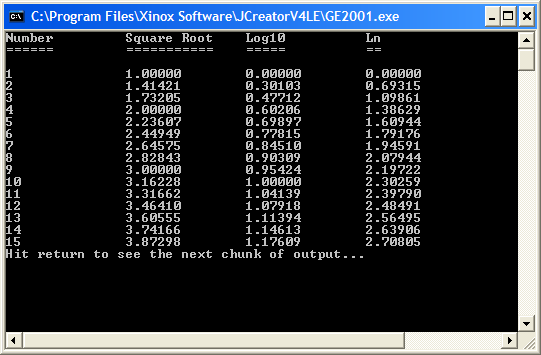
We will look further at the Math class methods in semester 2, but they are very useful for working out formulas, especially scientific and engineering based ones.

• The line is then displayed for a given value of i using the format string.

• The last thing inside the loop is a **simple if** statement which is used to divide the output into chunks of 15 lines each. The test expression here is

**(i%15==0 && i!=90)**

which checks to see if the value of i is exactly divisible by 15 and also is not 90 (the very last value of i). As long as these tests evaluate to true, then the program will put out a message to the user asking them to **hit return** to see the next chunk of output. The return character entered by the user will then be consumed by the call to **input.nextLine()** and then the table header will again be displayed. The effect of the code here is to stop the loops execution after every 15 lines of output and not continue until the user hits return, at which point the loop resumes and displays the table header along with the next 15 lines of output. The effect is as follows:



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

Click the **New File** icon on the JCreator IDE and save the file as **For2.java** in your Lab9 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.

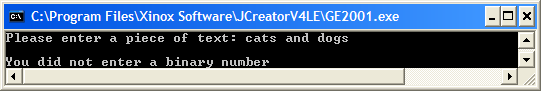
Once you are free from errors, run and **test the program –** note that in this case there is no user input whatsoever and the program will produce exactly the same output on every execution – very easy to test!

**Note that for the purposes of the final exam, the only mathematical calculations you need to be able to accomplish yourselves are the actual coding of given formulas and use of mathematical methods along with simple averages and percentages.**

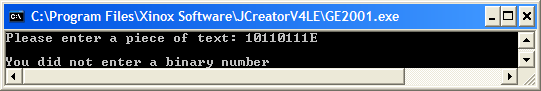
**Exercise 4**

Write a program that reads in a piece of text from the user and then uses a **for loop** to help determine whether the text supplied constitutes a binary number. In a binary number, all the characters are either 0 or 1. Try to maximize the efficiency of your code here so that, if a given character does not turn out to be a 0 or a 1, the loop should immediately exit. Your program should operate as indicated in the following sample screenshot:

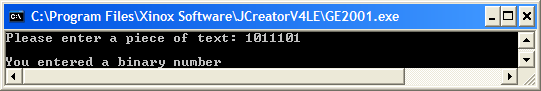
Run 1:



Run 2:



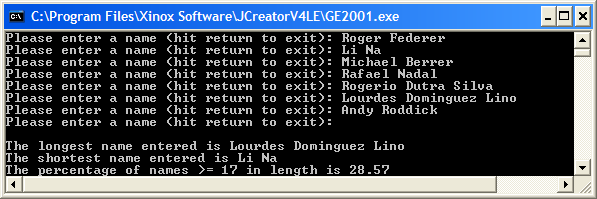
Run 3:



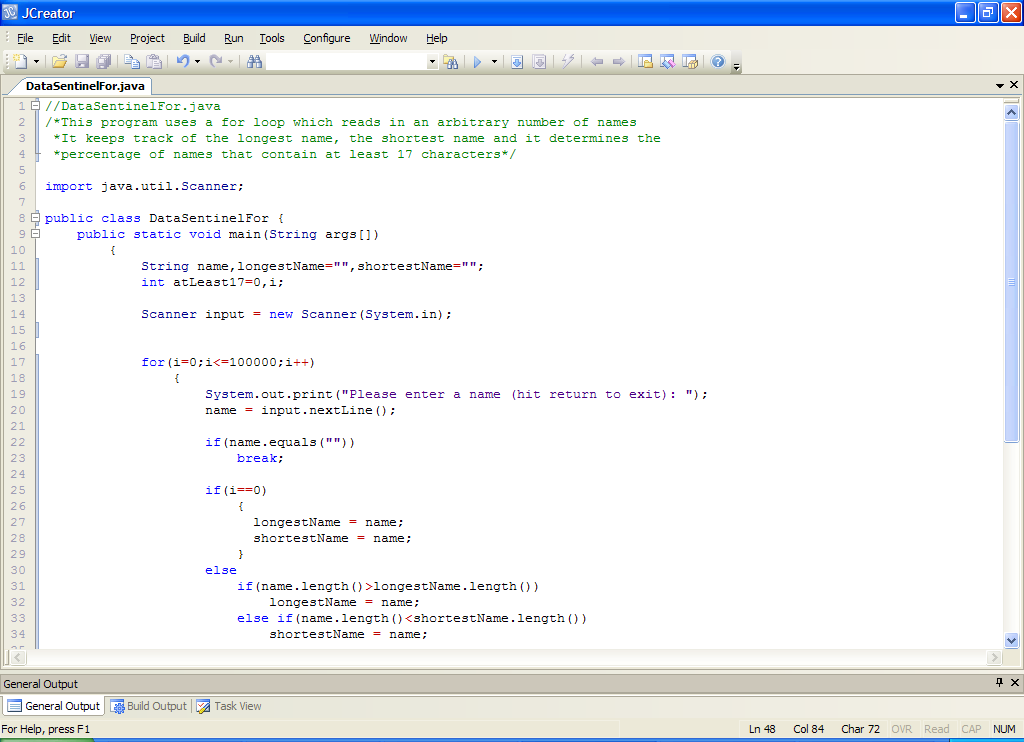
**Using a for loop in a Data-Sentinel Controlled Environment**

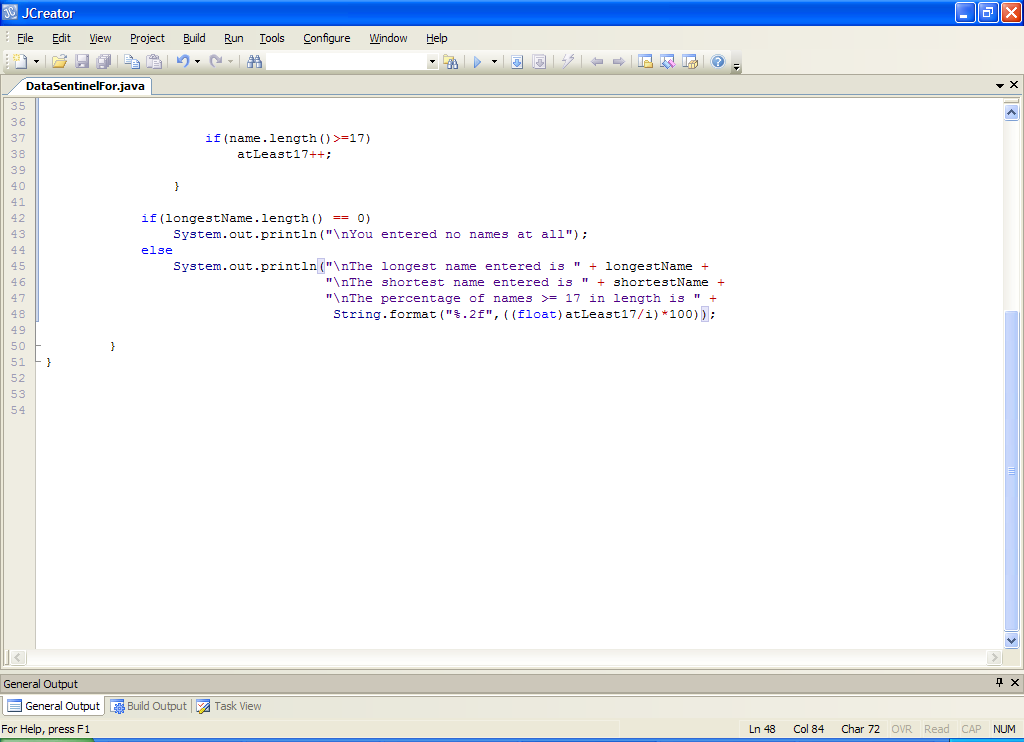
Although for loops were designed with counter-controlled iteration in mind, it is possible to use them in a data-sentinel controlled situation also.

**Aim:** To write a program using a for loop that reads in an arbitrary number of names. The program should determine the longest and shortest names entered along with the percentage of names that contained at least 17 characters. It would run as follows:



**Java Code:**





**Analysis of Program:**

• The program begins by initializing 2 String variables to **empty strings**. These are the variables that will store the longest and shortest names. There is also a counter variable for keeping track of the number of names that contain at least 17 characters. The loop variable i is also declared.

• Notice here that the test expression at the top of the while loop is

**i<=100000**

It may seem odd that the number on the right is so large, but remember here that we are trying to use the for loop to mimic the action of a data-sentinel controlled while loop. As we have no idea at compile time how many names will be entered by the user, we just try to make sure that we allow for sufficient names, hence the 100000. If we were to make this too small, for example, 10, and the user wanted to enter 20 names, then the loop would just stop after 10 iterations and there is nothing the user could do about it.

• Within the loop, the user is prompted for the name and this is read in. As this for loop is essentially data-sentinel controlled, we need to tell the user what the sentinel value is in the prompt. This is why we add **“hit return to exit”** to the prompt.

• The next code inside the loop checks to see if the user hit return when prompted for the name. If they did then the expression

**(name.equals(“”))**

will evaluate to true, in which case, we immediately want to exit the loop, hence the break statement.

• As long as the user enters something other than hitting return for the name, then the next block of code will execute. This code determines the longest and shortest name. It begins by checking to see if this was the first loop iteration with the test

**if (i == 0)**

if it was the first iteration of the loop then the user has just entered the very first name and so at this point the longest name and shortest name must both be that name. Hence the setting of longestName and shortestName to name.

If, however, it was any iteration after the first one, then the if section evaluates to false and the else section is executed. The else section here contains further tests in the form of a **nested else-if**. Here the first test is to check whether the name just entered is longer than the current longest name, with the code

**if(name.length()>longestName.length())**

**longestName = name;**

If the name turns out to be longer, then we reset the variable longestName to the value stored in name.

If the name does not turn out to be longer, the else if section executes which tests to see whether the name was shorter than the current shortest name, with the code

**else if(name.length()<shortestName.length())**

**shortestName = name;**

If the name turns out to be shorter, then we reset the variable shortestName to the value stored in name.

• The last piece of action within the for loop is to check whether the name entered contains at least 17 characters and, if it does, increment the value of **atLeast17**.

• Once the for loop finishes, there is an if-else statement. The if test here checks to see whether the value of longestName.length() is zero. If it is, then the user must have entered no names at all, so a message along these lines is displayed. Otherwise, the user must have entered at least one name, and so we display the details of the longest and shortest names along with the percentage of names containing at least 17 characters. Can you see why the formula for determining the percentage contains **type-casting** here? If in doubt, remove the type-casting when you are testing the program later and recompile and run the program.

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

Click the **New File** icon on the JCreator IDE and save the file as **DataSentinelFor.java** in your Lab9 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 fully** for a number name values. Also run it by entering no names at all to see what happens.

**Interchangeability of the for loop**

So we have just demonstrated how a for loop can be used in data-sentinel controlled situation. In fact, a for loop can always be used in place of a while or do-while loop and vice versa. It’s really a matter of personal preference as to which type of loop is used. However, the while loop remains the most widely used loop type by far in practice, used in all types of situations, whereas the for loop is used predominantly in counter-controlled environments, with the do-while the least used of the three, restricted normally to the likes of menu-type programs. The best you can do as a programmer is to try to get familiar with all three kinds of loop so that you are comfortable with using them all. If you are taking the OOP1 module in semester 2 you will get a lot more exposure to looping structures, especially the while and for loops.

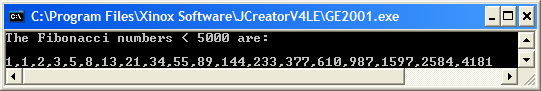
The next exercise gets you to use a for loop in a **task-controlled** environment.

**Exercise 5**

The sequence of **Fibonacci numbers** begins with the integers

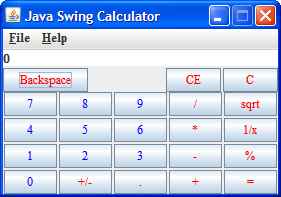
1, 1, 2, 3, 5, 8, 13, 21, ...

where each number after the first two is the sum of the two preceding numbers. Using the appropriate algorithm, write a program called **Exercise4.java** which will display all the Fibonacci numbers smaller than 5000. Note that the first two Fibonacci numbers will have to be displayed directly since neither has 2 predecessors. Your program should use a **for** loop for the iteration process here and use a **break** statement to exit when the terminating condition arises. It should run as indicated in the following sample screenshot:



**Java in the Real World**

Once again, we will have a look at an example of a real-world Java program. This one is a basic calculator application written by B. Hemanth and the details can be found at <http://www.javabeginner.com/java-swing/java-swing-calculator> . I have downloaded the source code also so you can have a look at it later but first of all open up the file **Calculator.java** in the **Calculator** folder and compile and run it. This file is an application rather than an applet so it is designed to run on the desktop.



Now just use the calculator for a while, check out some of its functionality to make sure it works as advertised.

Now, turn your attention back to the JCreator window and examine the code for the program. You can see that many classes are being imported for this application. We will have a few labs in the semester 2 OOP1 module based on creating basic Java GUIs so we will be using some of the components this calculator uses at that stage such as the **buttons** you see above.

You can see that the application declares a number of integer **constants** using the **final** keyword. On line 119 you can see a **for loop** being used to create the 9 numeric buttons of the GUI and another for loop on line 153 being used to set the font and foreground colour of all the buttons. This second for loop is using an **if-else** to decide which of the buttons will have blue foreground and which will have red.

Notice the for loop beginning on line 247. You can see a switch within an if within the for loop here. It’s good to see such code as it impresses on us that structures in Java can often be nested within other structures, especially as applications become more complex. Even within the switch itself at **case 17** there is a **simple if**!

Again, don’t worry at all about the overall code, there is no way you are expected to be able to understand it all. Much of the code involves the creation of the GUI itself and what is referred to as **event-handling** code. The main thing again is that you can recognize certain structures, keywords and operators and if you can make sense of certain parts of the code it is a bonus.