**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

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

**Practical 2 – Quick Review of Java Basics**

Hopefully you managed to get through the first lab sheet. This one continues our review of the Java basics where we will be re-examining **repetition-control structures**.

**The while Loop**

This is the **main looping structure** in Java. Although other loop types exist, all repetition can be carried out using only while loops.

There are 3 application areas of while loops:

* Data-sentinel controlled situations
* Counter-controlled situations
* Task-controlled situations

**Data-sentinel Controlled While Loops**

Here the loop is guarded by a **sentinel value** which means that the loop will continue to execute until the sentinel value is entered by the user. An example would be where you are reading in and maintaining a list of names from the user and you want the loop to execute until the user hits return on the keyboard when prompted for a name as follows:

**System.out.print(“Please enter a name (hit return to exit): ”);**

**name = input.nextLine();**

**while(!name.equals(“”))**

**{**

**allNames = allNames + name + “\n”;**

**System.out.print(“Please enter a name (hit return to exit): ”);**

**name = input.nextLine();**

**}**

The nature of the data-sentinel controlled while loop is that the user gets asked to enter a first value, before the loop starts. This first value entered is then compared to the sentinel value at the top of the loop. As long as the user hasn’t entered the sentinel value, the loop body will execute. At the end of the loop body the user will again be asked to enter a value. This process repeats as long as the sentinel value is not entered. Key to the data-sentinel controlled situation is that the user of the program knows what they need to enter in order to bring the loop to a halt, hence the “**hit return to exit**” message within the prompt. This type of loop is used in a situation where we don’t know how many times the loop will have to execute and it is up to the user to bring the loop to a halt.

**Counter-Controlled While Loops**

Here the loop iterates a certain, predictable, number of times, based on the value of a loop counter variable. An example of a situation where such a loop might be used would be where you wish to enter exactly 10 numbers and determine the largest and smallest number entered as follows:

**counter = 1;**

**while(counter <= 10)**

**{**

**System.out.print(“Please enter a number: ”);**

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

**if(counter == 1)**

**{**

**largest = number;**

**smallest = number;**

**}**

**else if(number>largest)**

**largest = number;**

**else if(number<smallest)**

**smallest = number;**

**counter++;**

**}**

The nature of the counter-controlled while loop is that a loop counter variable is set to some initial value before the loop starts (1 in this case). Then the value of the counter variable is compared with some final value (10 in this case). As long as the value of the counter variable is not greater than the final value, the loop will iterate. Each time it iterates, some processing occurs – in this instance the processing involves taking in a number from the user and then keeping track of the largest and smallest numbers entered via the **nested else-if** structure. Then, just before the loop body finishes, the counter variable gets incremented. This ensures the loop will eventually stop.

The code snippet above demonstrates the **increment operator** **++** which is **shorthand** for

counter = counter + 1;

There is also a **decrement operator --**

**Task-Controlled While Loops**

Here the loop process repeats until a certain task has been completed. The loop iterates an unpredictable number of times (like the data-sentinel case) and normally completes automatically (like the counter-controlled case). An example of using a task-controlled while loop would be where you want to determine the smallest positive integer whose cube exceeds 5000 as follows:

**number = 0;**

**while(number\*number\*number <= 5000)**

**{**

**number++;**

**}**

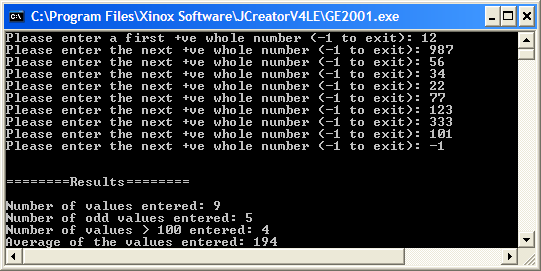
Every time the loop is entered, it first checks whether **number\*number\*number (number3)** does not exceed 5000. As long as it doesn’t, the variable number gets incremented.

**Creating a Folder for This Lab Sheet**

On the X: drive, in your OOP1Stuff folder, you should create a new folder called **Lab2** to store your work for this lab sheet.

**Exercise 1**

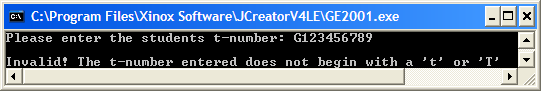
Write a Java program that reads in an arbitrary amount of positive whole numbers (-1 to exit). The program should keep track of the number of values entered and determine the average of the numbers entered to the nearest whole number. It should also keep track of the number of odd values entered along with the number of values entered that exceeded 100. Your program should run as indicated in the following screenshots:



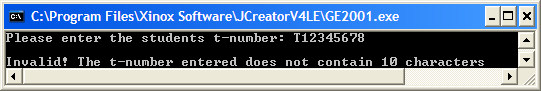
**Exercise 2**

Write a Java program that reads in the t-number of a student. A t-number should have exactly 10 characters in it and should begin with a lowercase or uppercase ‘t’. It should then contain exactly 9 digits after that. The program should determine whether or not the t-number entered by the user is valid and report the appropriate message as indicated in the following sample screenshots – make sure to **test** your program fully here.

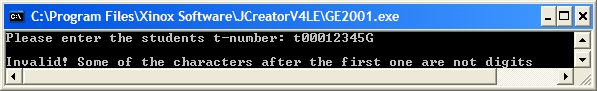
Run 1:



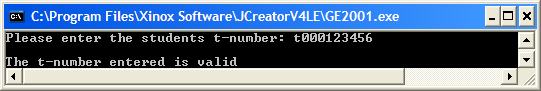
Run 2:



Run 3:

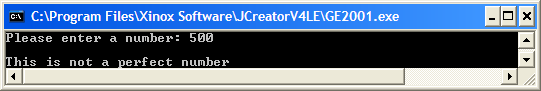


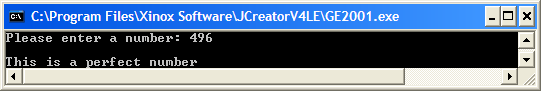
Run 4:



**Exercise 3**

In mathematics, a perfect number is a positive whole number whose positive factors (excluding the number itself) add up to that number. So, for example, 6 is a perfect number since its positive factors (the numbers that divide exactly into it) are 1, 2 and 3, which add to give 6. Write a Java program that gets the user to enter a number and determine whether or not it is perfect. Try to use a task-controlled while loop in your solution here whose tests make the program as efficient as possible. For example, the divisor should not go beyond half the value of the number (since any value greater than half a number cannot divide evenly into it and so cannot be a factor). There is another way to make the loop a little more efficient also, think about what it might be. In any case, try to solve the problem as best you can. It runs as follows:





**The do-while Loop**

An alternative to the while loop. The do-while loop is used for “at least once” repetition since the statements that make up the loops body will always occur at least once. This is because the loop test expression occurs at the end of the structure, whereas it is always at the top for a while loop. It can always be used in place of a while loop, with minor coding modifications. They are very often used for menu-based programs, where the menu gets displayed at least once but another example of using a do-while loop might be where you want the user to enter the age of a person and then do some simple validation on the age as follows:

**count = 0;**

**System.out.print(“Please enter the age: ”);**

**age = input.nextInt();**

**do {**

**if(count!=0)**

**{**

**System.out.print(“Invalid! Please re-enter the age: ”);**

**age = input.nextInt();**

**}**

**count++;**

**} while(age<0 || age>130);**

Here the user is prompted for the age. Then, within the do-while loop, a test is performed to see whether it is the first loop iteration. If it is, then the statements associated with the simple if are bypassed, the variable count will increment and then the loop test expression is evaluated. If the age entered was valid i.e. between 0 and 130 inclusive, then the loop will now exit, otherwise it will re-iterate and, because count is now non-zero, the “Invalid” message will appear, with the user given a chance to re-enter the age. This process will repeat until the user enters a valid age. We will spend a lot of time on validation routines later on in this semester so you will get a chance to see this type of algorithm again. The validation here is only partial, in that it assumes the user enters a whole number for the age. You will see later that for proper validation, we must assume the worst and take it the user could enter anything for the age e.g. a word etc.

**The for loop**

An alternative to the while loop. The for loop is synonymous with counter-controlled repetition, but we saw in semester 1 that it could easily be tweaked in order to work in data-sentinel controlled or task-controlled situations also. Programmers generally have a tendancy to use for loops for counter-controlled iteration rather than the while loop – it is considered more efficient. An example of where a for loop might be used would be to read in 10 numbers and then determine those numbers that are both multiples of 2 and 5. 10 would be a multiple of both 2 and 5 because both 2 and 5 divide evenly into 10.

**for(i=0;i<10;i++)**

**{**

**System.out.print(“Please enter a number: ”);**

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

**if(number%2==0 && number%5==0)**

**multiplesOf2And5 += number + “\n”;**

**}**

The for loop has 3 parts at the top of it. The first part **initializes** the loop counter variable (to 0 in this case – this happens only once), the second part is the loop **test expression** (in this case the loop counter is compared with 10), the last part is the **increment** of the loop counter, which actually happens *after* the loop body statements have executed. Here a number is read in each time the loop iterates and then a test is performed to see if the number is a multiple of both 2 and 5. If it is then it is appended to a String variable called multiplesOf2And5, to be displayed later once the loop finishes.

Note the use of the **+=** operator above also. This is called the **arithmetic assignment operator** **for addition**, although here there is no arithmetic, it is used to join strings together instead. It is shorthand for

**multiplesOf2And5 = multiplesOf2And5 + number + “\n”;**

In arithmetic, there are other variants of it for each of the arithmetic operators e.g.

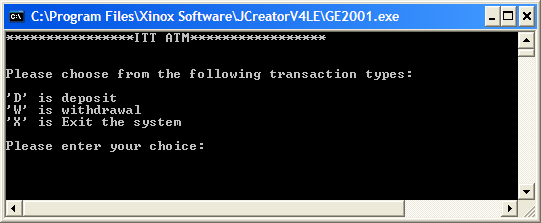
**a -= b;**

is equivalent to

**a = a – b;**

**Exercise 4**

Write a program that uses a **do-while** loop to simulate the actions of an ATM machine. It begins by presenting the user with a menu of 3 transaction options as indicated in the screen capture below:

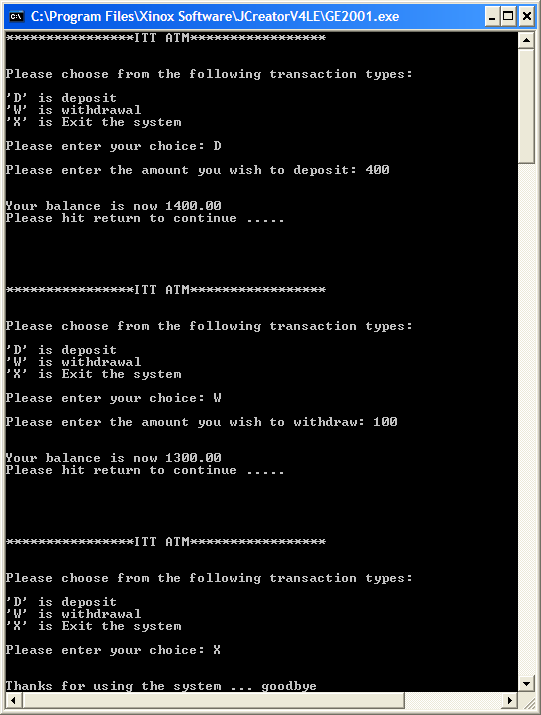


The user then enters their choice as an **uppercase letter.**

Once a valid choice has been entered, the user can deposit, withdraw or exit from the system. They can carry out as many deposits or withdrawals as they like and each one updates their balance and then displays the new balance to the screen to 2 decimal places. You can assume that the user starts off with an initial balance of €1000 in their account.

When the user selects the exit option then they receive a “goodbye” message.

The program should run as indicated in the following sample screenshot, which shows 3 iterations of the do-while loop. You shoulduse a **switch** statement within the do-while loop for this program for the decision-making (if you are a newcomer and haven’t seen switch before, have a look at lab 5 of the “Introduction to Programming” module to see how it works) and try to use the **arithmetic assignment operators** wherever possible.



**Exercise 5**

Write a Java program that reads in exactly 5 words using a **for** loop. The program should determine the average number of vowels per word along with the average number of consonants per word and display these to 2 decimal places (note that you’ll need **another for loop** within the main for loop for this processing). You can take it that all the words entered will be valid and will only contain letters. It should also determine the longest word entered. Note that your program should run as indicated in the following sample screenshot.

