**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

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

**Practical 6 – User-defined Methods**

In this lab we continue to explore methods. Up until now we have looked at programs containing just one user-defined method. This time we will look at the notion of problem-solving in conjunction with methods and see how a larger problem can be split up into a number of smaller tasks through the use of several methods.

**Problem-solving with Methods**

It is a while since we have discussed **problem-solving** in any detail but hopefully you have been able to apply this section of last semesters module as you have been going through this one. Remember that **problem-solving is generic** but **programming languages are specific**. Writing a valid solution to a problem in **pseudocode** shows that you have understood the problem and have devised and carried out the plan to solve the problem. Hopefully, you will also have assessed the result and documented your solution. These are the 5 keys to successful problem solving.

Last semester we never discussed user-defined methods and their design. Remember that problem-solving is all about taking a larger problem and breaking it up into a set of smaller, more manageable sub-problems. You may also remember that the term **modularisation** refers to pretty much the same idea – using many small, self-contained, modules (methods) in order to carry out the larger programming problem. In this way, your program can be modularised into a collection of methods, user-defined and pre-defined, which describe very succinctly what the program is doing. Also, if the program repeats the same task a number of times in the program, we **can use the same method several times** without having to repeat large blocks of code which would be very inefficient.

It is obvious, therefore, that **using methods should be a key part of developing good programming algorithms**.

In terms of problem-solving with methods there are **2 basic rules**:

• each method should be **small**, **self-contained** and perform just **one major task**

• **use** **refinements** to break a user-defined method that is too complicated down into several smaller tasks, each one implemented by its own method.

Now the process of writing solutions to programming problems becomes:

**Draft 1 of Pseudocode** - give a **very high level description** of the steps to be carried out by the program

**Draft 2 of Pseudocode** – includes refinements to draft 1

**Continue Refining***,* creating further drafts until the main() body is complete

**Identify any actions within the final draft of pseudocode that can be carried out by using user-defined methods and repeat the refinement process for each method used.**

**An Example of Problem-Solving with Methods**

Write a program which will read in the following details of an arbitrary number of customers:

* account number (let this be the sentinel variable)
* opening balance
* total debits
* total credits
* credit limit.

Once these details have been entered for any given customer their closing balance will be determined and checked to determine if their credit limit has been exceeded or not.

Then, on the one screen, display the customers account number, credit limit, closing balance and whether or not their credit limit has been exceeded.

When the user has stopped entering details, they should then receive a farewell message

**Pseudocode**

Firstly after reading the problem specification, we note that information for an **arbitrary** number of customers has to be input. Therefore we will use a **data sentinel-controlled** while loop.

We proceed with a first draft of pseudocode and then refine as necessary.

**Draft 1 of Pseudocode**

*prompt for account number*

*read in account number*

*while sentinel value has not been entered for the account number*

*prompt for balance*

*read in account balance*

*prompt for account debits*

*read in account debits*

*prompt for account credits*

*read in account credits*

*prompt for credit limit*

*read in credit limit*

*determine if credit limit has been exceeded*

*display the account number, closing balance, credit limit and whether of not the*

*credit limit was exceeded*

*prompt for next account number*

*read in next account number*

*endwhile*

*display farewell message*

**Refinements to draft 1 of Pseudocode**

The action *while sentinel value has not been entered for the account number*

can be refined as follows:

*while (accountNumber ≠ “”)*

Note here that I have decided that the accountNumber will be stored as a String rather than as a numeric. Therefore I have chosen empty string (“”) as the sentinel value. If you had decided to use a numeric type for accountNumber, then you would choose a sentinel value such as -1 (any negative value would be fine).

The action *determine if credit limit has been exceeded*

can be refined as follows:

*calculate closing balance*

*if closing balance < - credit limit*

*display “Credit Limit exceeded”*

*else*

*display “Credit Limit not exceeded”*

Now **create draft 2 by substituting refinements into draft 1**

**Draft 2 of Pseudocode**

*prompt for account number*

*read in account number*

*while (accountNumber ≠ “”)*

*prompt for account balance*

*read in account balance*

*prompt for account debits*

*read in account debits*

*prompt for account credits*

*read in account credits*

*prompt for credit limit*

*read in credit limit*

*calculate closing balance*

*if closing balance < - credit limit*

*display “Credit Limit exceeded”*

*else*

*display “Credit Limit not exceeded”*

*display the account number, closing balance, credit limit and whether of not the*

*credit limit was exceeded*

*prompt for next account number*

*read in next account number*

*endwhile*

*display farewell message*

**Refinements to draft 2 of Pseudocode**

The action *calculate closing balance*

can be refined as follows:

*set closing balance to account balance – debits + credits*

Now **create draft 3 by substituting refinements into draft 2**

**Draft 3 of Pseudocode**

*prompt for account number*

*read in account number*

*while (accountNumber ≠ “”)*

*prompt for account balance*

*read in account balance*

*prompt for account debits*

*read in account debits*

*prompt for account credits*

*read in account credits*

*prompt for credit limit*

*read in credit limit*

*set closing balance to account balance – debits + credits*

*if closing balance < - credit limit*

*display “Credit Limit exceeded”*

*else*

*display “Credit Limit not exceeded”*

*display the account number, closing balance, credit limit and whether of not the*

*credit limit was exceeded*

*prompt for next account number*

*read in next account number*

*endwhile*

*display farewell message*

**No further refinements are necessary** for the main() method in this case.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Our next consideration here is to examine the final pseudocode draft for main() above and see whether any of the tasks can be carried out with **user-defined methods**.

In fact, all the prompting for and reading in of details can be carried out using small self-contained methods. Also the checking of the credit-limit and displaying of the account number, closing balance, credit limit and whether the credit-limit was exceeded can be carried out by another user-defined method. In total, therefore, we could have 6 user-defined methods here.

Now we **design the methods** using the following technique. As an example here, we will only look at designing the one that checks the credit limit and displays some final results.

● What should we **call** the method? It must have a meaningful name in the context of the program and what it is meant to do. So here a good name might be **checkCreditLimit**()

● What **arguments** does this method need to take? To answer this part you must know exactly what the method is supposed to do. To do this look at the existing final draft of pseudocode from main() at the part that applies for this method – it says

*set closing balance to account balance – debits + credits*

*if closing balance < - credit limit*

*display “Credit Limit exceeded”*

*else*

*display “Credit Limit not exceeded”*

*display the account number, closing balance, credit limit and whether of not the*

*credit limit was exceeded*

So we see that this method must be given the **account balance**, **debits** and **credits** – so that it can calculate the closing balance. It must also be given the **credit limit** so that it can be compared with the closing balance. Finally, it must be given the **account number** since it also aims to display this value.

Therefore this **method needs to take 5 arguments**.

● What will this method return? To answer this part you must examine what the method is doing. Here the method is to perform a calculation and then perform a comparison and then display some results to the screen. It does not need to give back any result to the method calling it however. Therefore this method **returns nothing**.

**Draft 1 of the Pseudocode** for this method is

*checkCreditLimit(account number, starting balance, debits, credits, credit limit)*

*set closing balance to account balance – debits + credits*

*if closing balance < - credit limit*

*display “Credit Limit exceeded”*

*else*

*display “Credit Limit not exceeded”*

*display the account number, closing balance, credit limit and whether of not the*

*credit limit was exceeded*

If we wanted to be picky here we could say that **this method could really be broken down further into two separate methods**. One which only does the actual checking to see if the credit limit was exceeded and another called displayResults() which only focuses on displaying the final outputs to the screen but which could call checkCreditLimit() to decide which message of “exceeded” or “not exceeded” appears on the display screen. I will get you to do this as an exercise shortly.

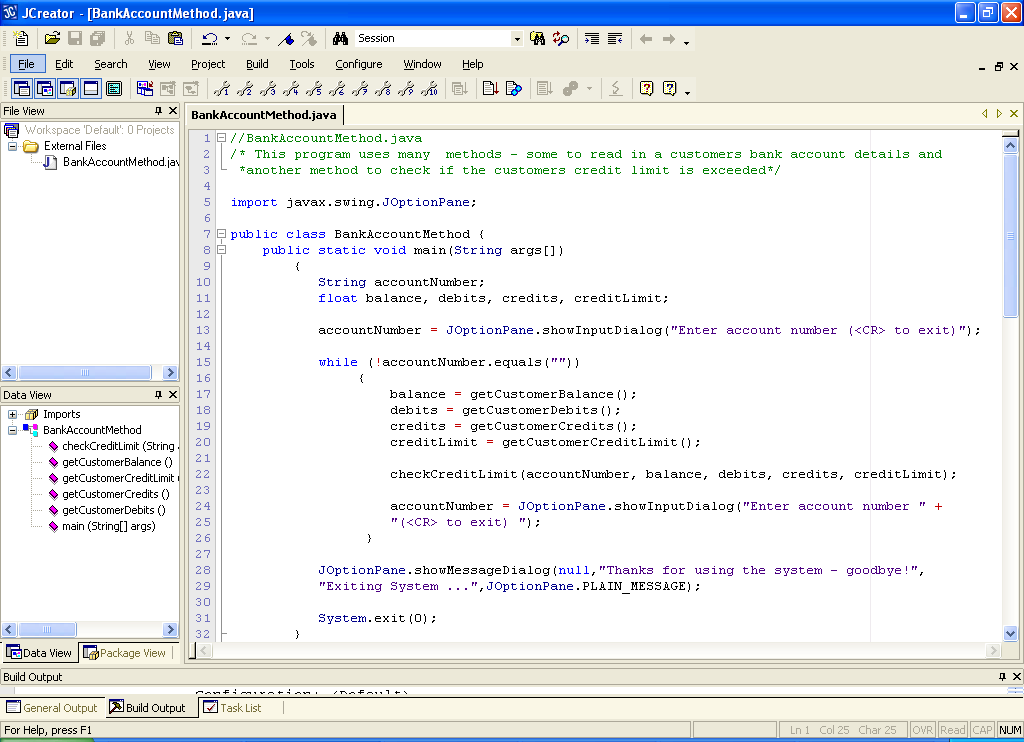
For now, however, we will say that no further refinements are necessary for the method checkCreditLimit ()

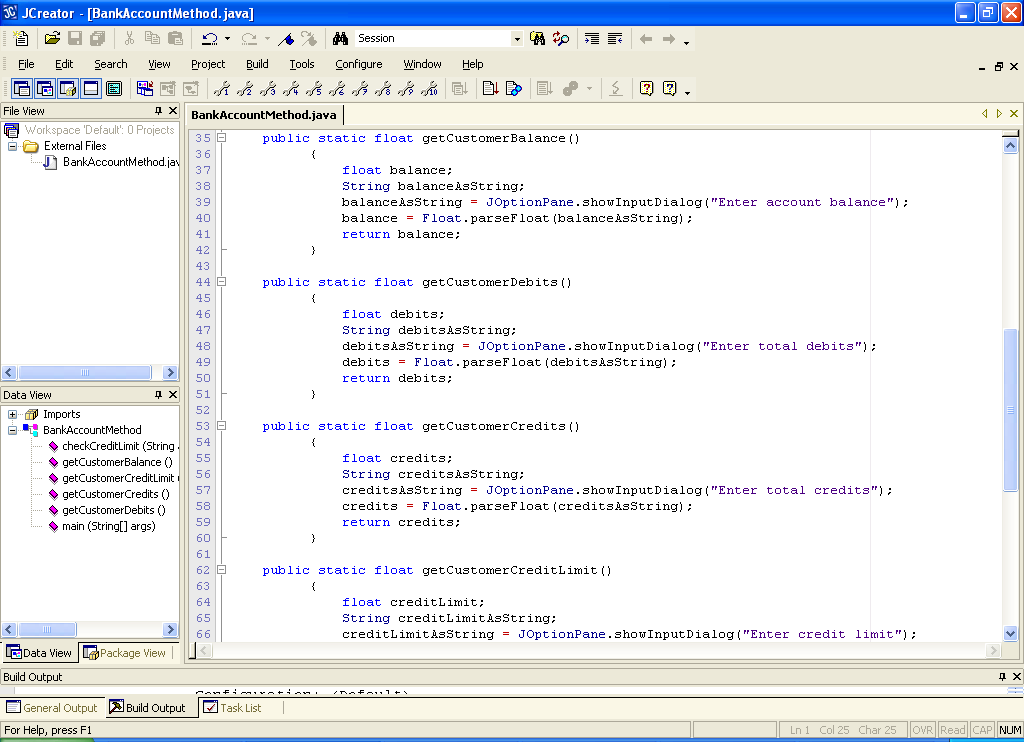
We can now proceed to write the program. This is our next sample program.

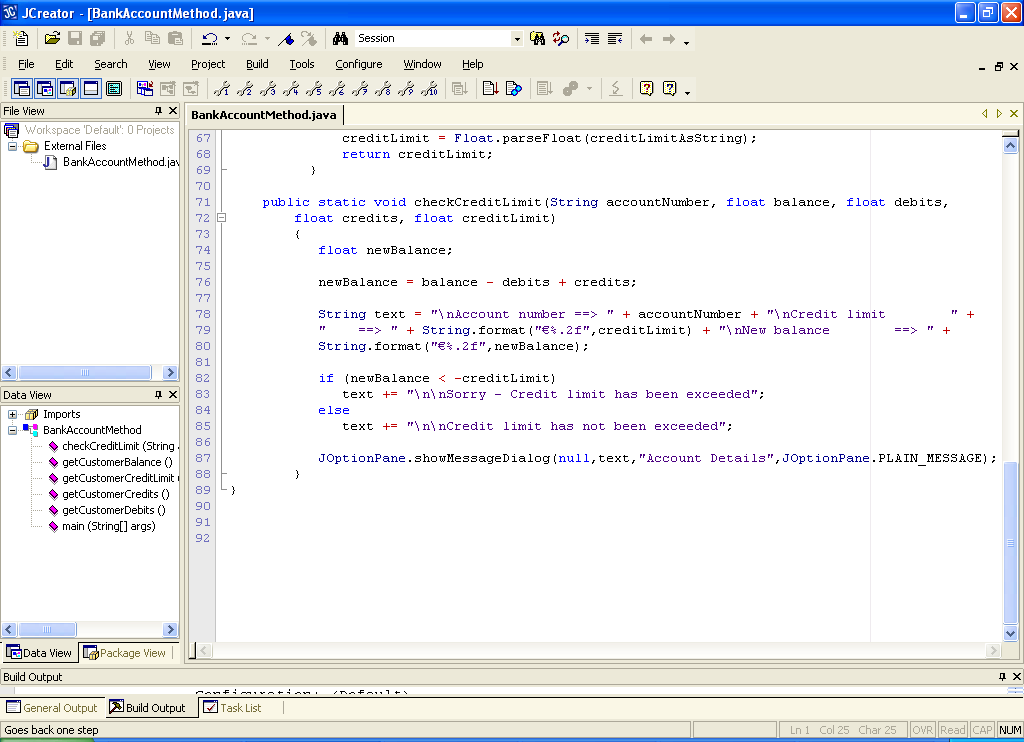
**Using Many User-Defined Methods**

**Aim:** We wish to write a program that contains several user-defined methods. Some of these will read in specific details about a bank account holders account and another will be used to check whether or not the account holder’s credit limit has been exceeded.

**Java Code**:





****

**Analysis of Program:**

● In this program the user is first asked to enter the customers account number which is read in and compared to the value empty string i.e. “”. A data-sentinel controlled while loop is in control of the iteration process in the program and should the value “” be entered for account number, then the while loop will cease to execute. Entering any value other than “” means that the loop body is entered and the following user-defined methods are called consecutively:

balance = getCustomerBalance();

debits = getCustomerDebits();

credits = getCustomerCredits();

creditLimit = getCustomerCreditLimit();

● the first statement above calls the getCustomerBalance() method which executes the following code:

public static float getCustomerBalance()

{

float balance;

String balanceAsString;

balanceAsString = JOptionPane.showInputDialog("Enter account balance");

balance = Float.parseFloat(balanceAsString);

return balance;

}

so we see that the user is asked to input the customers balance which is read in as a String from the input dialog and then converted to a float. It is then returned to the main() method via a return statement. In the main() method, the value returned from this method is stored in the variable **balance**.

● the second, third and fourth statements operate in an identical manner to return the customers debits, credits and credit limit to the main() method for storage in the appropriate variables.

Note that each of these 4 methods **take no arguments**. This is perfectly normal, of course. There are thousands of methods within the Java API that take no arguments. When a method takes no arguments, then there is nothing within the parentheses (not even **void**).

● Once all this information has been entered, it is passed on to the checkCreditLimit() method as follows along with the customers account number already entered

checkCreditLimit(accountNumber, balance, debits, credits, creditLimit);

This method is at the heart of the program since it is the one which will determine and output whether or not the customer has exceeded his/her credit limit. A new balance is calculated based on the current balance, debits and credits and this is then compared with the credit limit of the customer. An **if-else** statement is used within the method to determine whether or not the customer is within the credit limit.

**if (newBalance < -creditLimit)**

note the code here is assuming the creditLimit has been entered as a positive value (the norm). Hence the minus in front of the variable creditLimit in the expression.

● This program is nice in the sense that it brings together many of the structures you have seen so far. We have a data-sentinel controlled while loop in the program (can you name the other types of while loop?), an if-else statement and of course user-defined methods along with other bits like an import statement, comments, predefined methods and arithmetic and relational operators.

● One important thing to note about this program is that, because practically all the user input processing has been delegated to separate user-defined methods, the **main() method is much shorter** than it would otherwise be. The presence of the user-defined method calls **makes the logic within the main() method much clearer** and is another big advantage of them.

**Organising your Work**

You should have a folder under X: called OOP1Stuff created. This time, create a folder called **Lab6** within the folder 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 **BankAccountMethod.java** in your Lab6 folder. Now, for practice, type in the code for the program above. Remember to use the **copy and paste** facility wherever possible to speed up your coding time.

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.

**Exercise 1**

In the program above, the customers account number was read in using ordinary “sequential” code on two occasions - before the start of the while loop and at the end of the while loop body. It would be **more efficient** in this case to use a user-defined method which achieves the same result. Save the program above as **Exercise1.java** and make the modifications so that it contains another user-defined method called **getCustomerAccountNumber**() which asks the user to input their account number and returns it to main() for storage in the **accountNumber** variable. Obviously you need to call this method twice as already stated.

**Exercise 2**

It was mentioned earlier that the checkCreditLimit() method could be split up into 2 separate methods. One will still be called checkCreditLimit() but the other will be called displayResults(). The checkCreditLimit() method will only check to see whether the credit limit has been exceeded. If it has, then the method will return the **boolean** value true to indicate this. If it has not, it will return the boolean value false instead.

The displayResults() method will only display the end results to the screen i.e. the closing balance, account number, credit limit and whether or not the credit limit was exceeded or not. Note that this method should call the checkCreditLimit()method in order to display the appropriate message in relation to the credit limit.

You should just take the **Exercise1.java** and save it as **Exercise2.java** and make the necessary modifications. Crucially, the program should run exactly the same as before.

**Variable Scope**

Recall the method definition header for the checkCreditLimit() method from the program above

public static void checkCreditLimit(String accountNumber, float balance, float debits,

float credits, float creditLimit)

You see that the name of the arguments are accountNumber, balance, debits etc.

These are **exactly the same** as the variable names defined within main() earlier. So how come there is no conflict?

There is a rule is Java stating that **you can only declare a given variable once within a method**. However, here there are 2 methods involved and the variables defined in each are **invisible** to the other one.

This is because **method arguments** and **variables defined within a method** are “**local variables**” which **cannot be seen outside of the method** they are defined in. They are said to have **local (block) scope**. This means that we can create variables in different methods with exactly the same name, if we wish to. However, it is considered **better programming practice to avoid duplicating variable names** since it **avoids ambiguity** when discussing a program, for example.

When it comes to defining methods, it is also important to know that coding a method argument is like declaring a variable within that method. From the point of view of variable scope this means that it would be **illegal to declare a variable within the method body with exactly the same name as one of the method arguments** e.g.

public static void testMethod(String text)

{

String text;

// rest of method definition

}

Would cause a **syntax error** as there is a conflict between the argument text and the local variable text.

It is interesting to note that it is **legal to have a method with exactly the same name as one of its arguments or local variables** e.g.

public static float average(float a, float b, float c)

{

float average;

// rest of method definition

}

is legal and is actually seen quite often.

**Exercise 3**

A menu-driven program called **Exercise3.java** is required which will give the user 4 choices as follows:

1. 1. Calculate the area of a cylinder
2. 2. Calculate the area of a semicircle
3. 3. Calculate the area of a triangle
4. 4. Quit

Your program should use **user-defined** **methods** for displaying the menu and dealing with the necessary calculations. Therefore you will have 4 user-defined methods in total here. The 3 methods that perform the calculations will also include the code for prompting and reading in the values from the user as well as displaying the results of the calculations. Therefore **all 4 methods will have a return type of** **void**.

The relevant formulae in this case are:

Area of cylinder = πr2h

Area of semi-circle = (πr2)/2

Area of triangle = (bh)/2

where r is the radius, h is the height and b is the length of the triangles base and π = 3.14

When the user has finished using the system, a farewell message should be issued.

You should:

(a) Write out the 1st draft of the pseudocode in this case

(b) Make any refinements to this draft and continue refining as necessary.

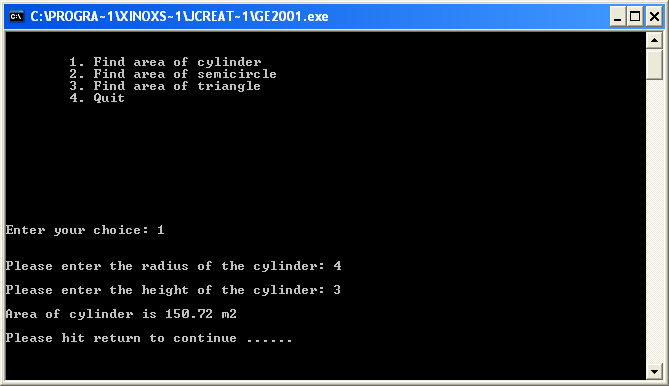
(c) Then proceed to take each method and make the necessary refinements to each one.

(d) Finally write the program as a **console application**

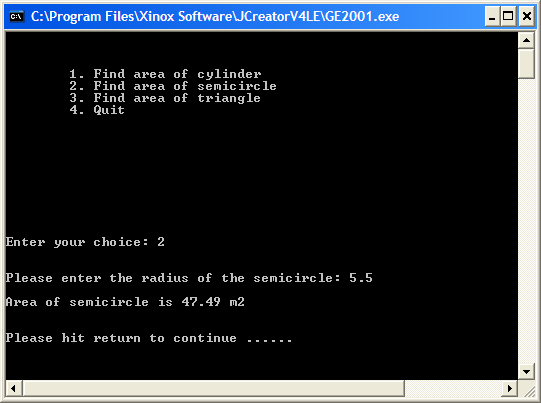
You should use a **switch** statement within your main() for the decision-making process and a **do-while** loop for the main iteration process. The program should issue an “invalid” message repetitively should the user enter an invalid choice value.

Your program should run as indicated in the following sample screenshots:

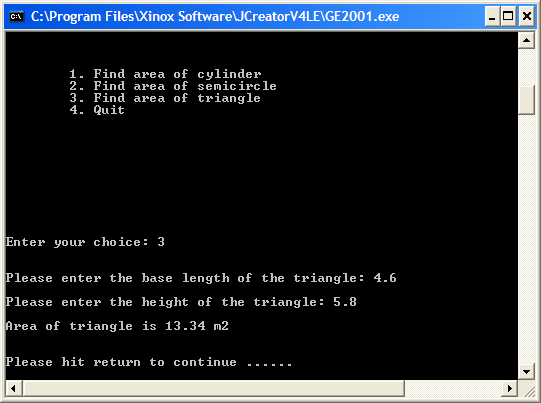
**Entering choice 1:**



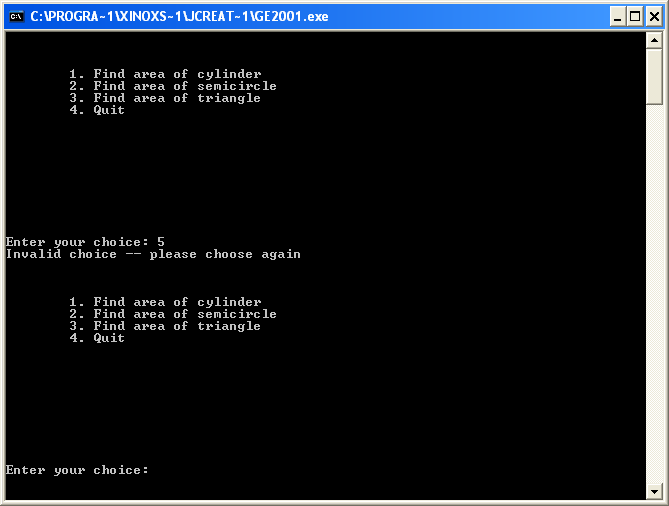
**Entering choice 2:**



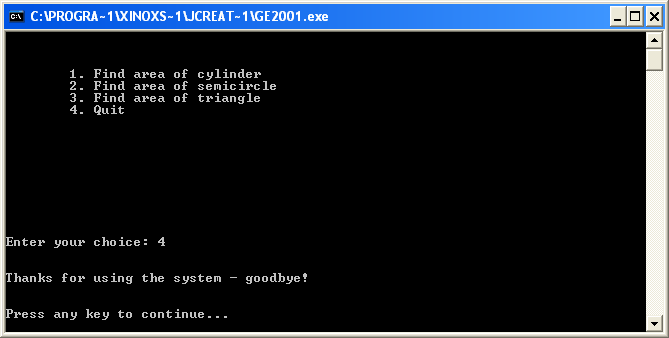
**Entering choice 3:**



**Entering an invalid choice:**

****

**Quitting the System:**

****