**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

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

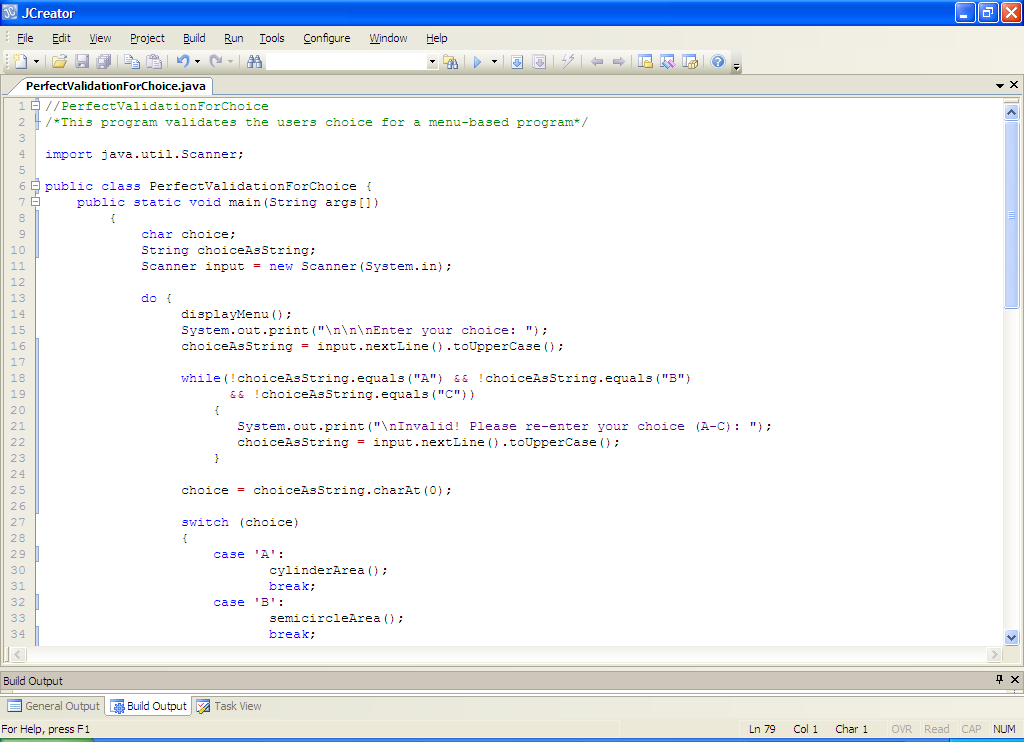
**Practical 9 – Input Validation**

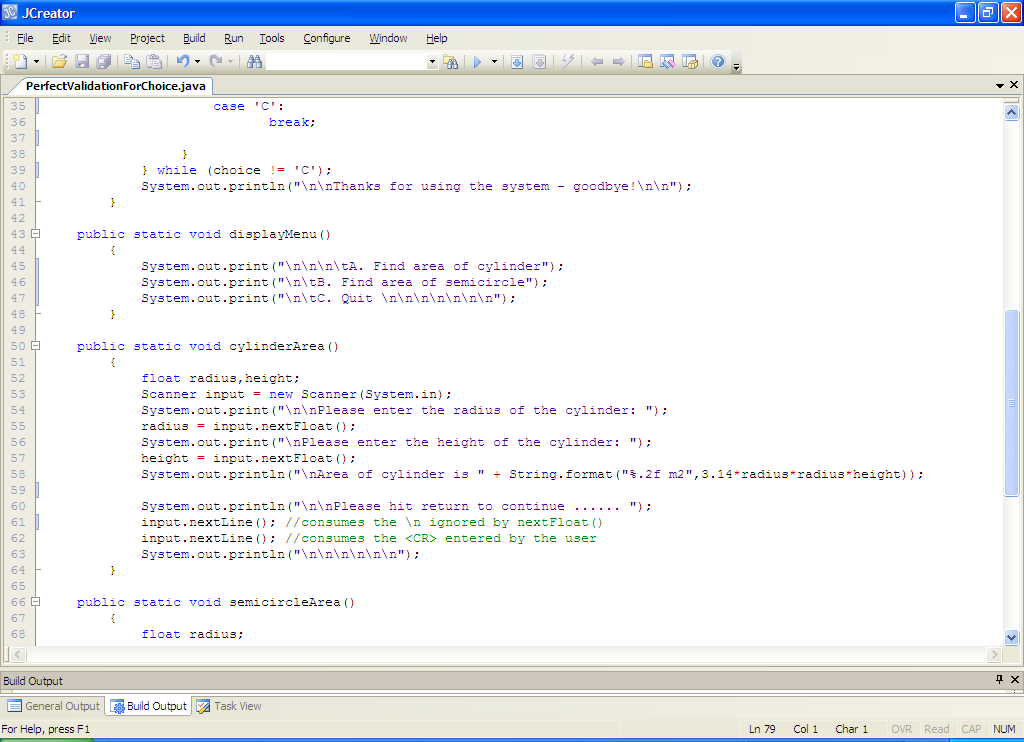
In this lab sheet we continue our examination of input validation algorithms.

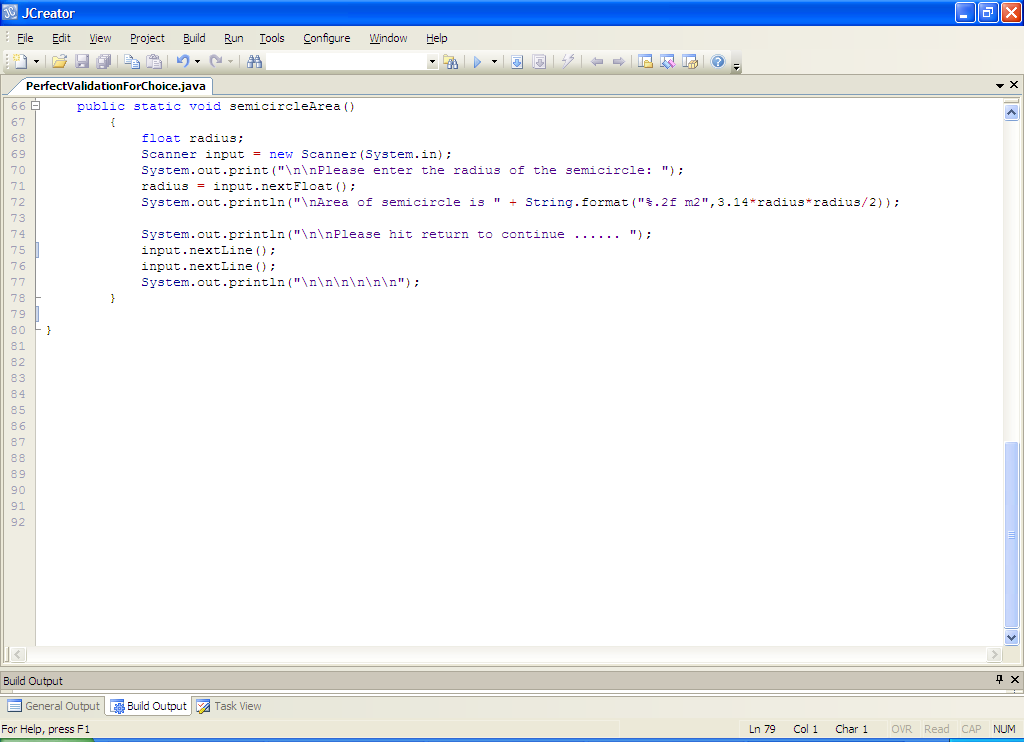
**Validating the Users Choice in a Menu-based Program**

**Aim:** We wish to display a set of menu options to the user and use a validation routine to make sure the users choice is validated perfectly.

**Java Code:**







**Analysis of program:**

• The main loop is this program is a **do-while** loop which first of all displays a set of menu options to the user with the displayMenu() method. Following this, the program reads in the users choice as a string using nextLine(). Notice the code here:

**choiceAsString = input.nextLine().toUpperCase();**

what happens is that the user choice is read in through nextLine() and then it is immediately converted to uppercase form through the method toUpperCase(). This type of cascading of method calls is legal and makes for shorter code, but you are always welcome to do it as follows instead:

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

**choiceAsString = choiceAsString.toUpperCase();**

The idea of calling toUpperCase() here is to allow the program accept the lowercase versions of the letters A-C along with the uppercase versions.

• Next up we have the validation loop for the users choice:

**while(!choiceAsString.equals("A") && !choiceAsString.equals("B")**

**&& !choiceAsString.equals("C"))**

**{**

**System.out.print("\nInvalid! Please re-enter your choice (A-C): ");**

**choiceAsString = input.nextLine().toUpperCase();**

**}**

This is a **data-sentinel controlled while loop** that keeps going as long as the users choice is not “A”, “B” or “C” (although they could have entered “a”, “b” or “c” also). If it is none of these then the user gets asked to re-enter the choice. So the user can never move on to the next part of the program until they enter a valid choice value. This validation routine also catches the **empty-string** possibility, as you will see later when you test the program. Therefore it works perfectly to validate the users choice.

• Following the validation loop, the users choice is converted to a char. This is done here because we are using a **switch** and this only works with int or char.

• The switch has 3 cases, corresponding to the 3 valid choices. If the user selects either “A” or “B” then the corresponding method is called to carry out the task required. Recall the use of the break within the switch to prevent execution rolling on to the next case. If the user enters “C” as their choice then the do-while loop will stop.

• The 3 user-defined methods follow the main(). All 3 are **void methods** in that they return nothing to main() when they are called. Also, none of the 3 take any arguments. The code

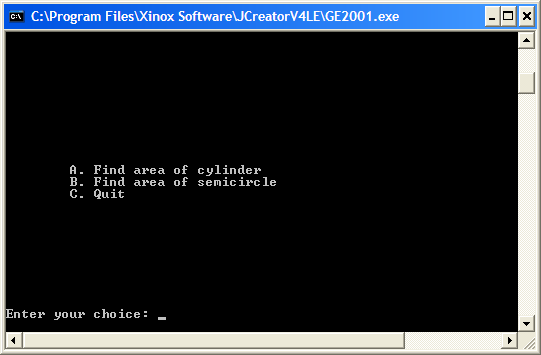
**input.nextLine();**

**input.nextLine();**

**System.out.println("\n\n\n\n\n\n");**

might look unusual, but in this program we are using Scanner and, because the newline character is always left in the keyboard buffer after using nextFloat(), we must consume this character first of all. Then we need a second call to nextLine() so that the program will halt after the “Please hit return to continue…” message is issued.

The display of a string of newline characters after this is just cosmetic. It is an attempt to “clear the screen” so that the next set of menu-options appear on what looks like a cleared screen as follows:



**Organising your Work**

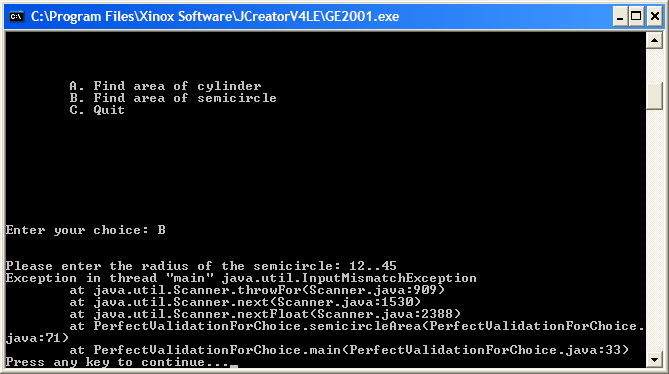
Within OOP1Stuff create a folder called **Lab9** to save your work from this lab session.

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

Click the **New File** icon on the JCreator IDE and save the file as **PerfectValidationForChoice.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. You will see that the validation of the users choice is perfect. You can also see that if you enter any non-numbers for the radius or height, the program will crash.

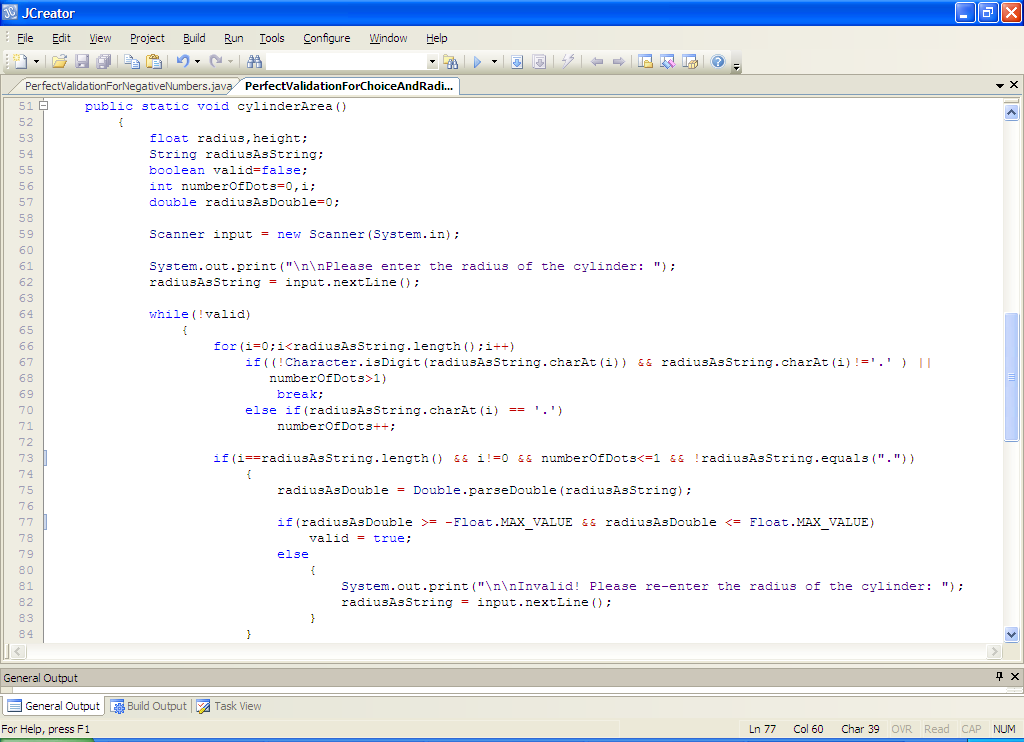


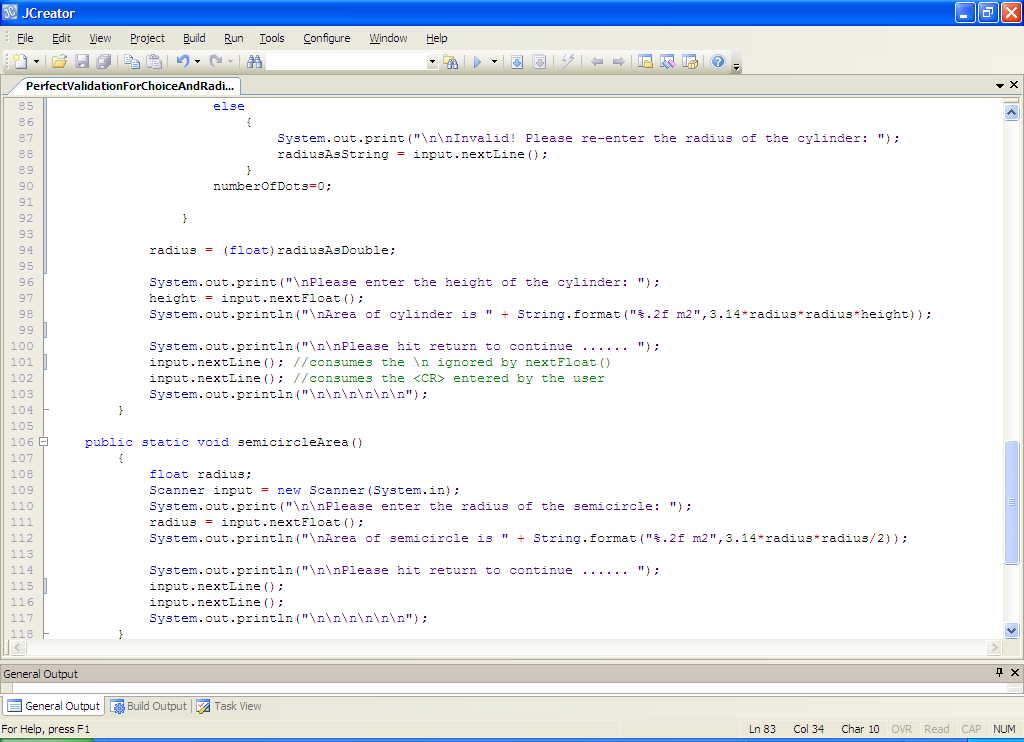
We will sort this out in the next program, but the validation will be much more involved than the users choice.

**Validating Floating-Point Numbers**

**Aim:** We wish to display a set of menu options to the user and use a validation routine to make sure the users choice is validated perfectly. If the user selects choice “A”, they are asked to enter the radius of a cylinder, which is stored in a float variable. We wish to validate this perfectly also.

**Java Code:**





**Analysis of program:**

• The rest of the program is identical to the original version so we just look at the **cylinderArea**() method here, as this is where the validation takes place.

Although the radius is a numeric quantity, we must read it in as a string in order to validate it, hence the radiusAsString variable. You will also see a **boolean** variable called **valid**, initialized to false. As mentioned in the last lab sheet, this variable will act as a flag to enable us to initially get into the validation loop to check whether the radius is valid and eventually get out of it, when the value of valid is set to true. This only happens when the radius has been proven to be valid. The method also declares a variable called **numberOfDots** and initializes it to zero. This variable is part of our validation routine – basically we only allow zero or one dot in a valid floating-point number. Anything more than this and we know we are dealing with bad input.

• The radius is read in as a string and then the validation loop begins, checking to see if the value of the variable valid is false. If it is (it will be to begin with) then the loop gets entered.

• The first part of the validation here involves the code:

**for(i=0;i<radiusAsString.length();i++) if((!Character.isDigit(radiusAsString.charAt(i)) && radiusAsString.charAt(i)!='.' )**

**|| numberOfDots>1)**

**break;**

**else if(radiusAsString.charAt(i) == '.')**

**numberOfDots++;**

the for loop here will iterate a maximum of **radiusAsString.length()** times, but will exit early if it turns out that the character under consideration is neither a digit nor a dot. Also should it turn out that more than 1 dot has been encountered, the for loop will exit early via the break. If it turns out that the character was a dot (and the number of dots encountered isn’t greater than 1) the value of numberOfDots gets incremented.

• When the for loop has completed, it either exited early or else it iterated radiusAsString.length() times. If it exited early, then we know for certain that the value inputted was bad so next we have the code:

**if(i==radiusAsString.length() && i!=0 && numberOfDots<=1**

**&& !radiusAsString.equals("."))**

**{**

**//do other processing**

**}**

**else**

**{**

**System.out.print("\n\nInvalid! Please re-enter the radius of the cylinder: ");**

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

**}**

if it turns out that i equals radiusAsString.length() then we know that the for loop performed the maximum number of iterations possible, so the value supplied must only have contained digits and dots. However, we must also handle the **possibility that the empty string was entered**, as we did in the last lab sheet. If this was the case, the value of i would be zero, so we just check for that possibility. Next, we must handle **the possibility that the last character of the value entered was a second dot**. If this happened, the loop would not have exited early (as it was the last character anyhow) but the value of **numberOfDots** would have increased to 2. So we test for this possibility with the next part of the if test above. Finally, the user might have entered just a single dot character. Again, this is an exceptional case that we need to handle specifically if we don’t want our program to crash so the last part of the if test handles this.

If the tests above result in a true outcome, then we can be certain that the value we entered was a valid floating-point number. Otherwise, there was something wrong with the input and the user gets the “Invalid!” message and is asked to re-enter.

• If it passed the test above, the next thing is to **convert the radius from a string to a double** value.

**radiusAsDouble = Double.parseDouble(radiusAsString);**

**if(radiusAsDouble >= -Float.MAX\_VALUE && radiusAsDouble <= Float.MAX\_VALUE)**

**valid = true;**

**else**

**{**

**System.out.print("\n\nInvalid! Please re-enter the radius of the cylinder: ");**

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

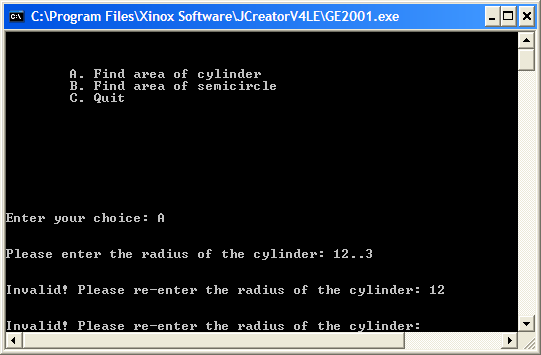
**}**

We do this here, for a similar reason as we did for the integer in the last lab sheet, because here the radius variable is a **single-precision** float variable, which has a minimum value of around -3.40282347E+38F and a maximum value of around 3.40282347E+38F. These are represented as **constants** in the Float class of the Java API using -**Float.MAX\_VALUE** and **Float.MAX\_VALUE**. If the user enters a value beyond these limits the program would crash if we tried to store the value in a float variable like radius. If the users value is within these limits, we know that the value supplied is valid in every way, and therefore we can now set the value of variable valid to true. This causes the validation loop to exit. If it turns out that the value entered is beyond the limits for float variables, the “Invalid!” message gets displayed and the user asked to re-enter.

• At the very end of the validation loop is the code:

**numberOfDots=0;**

which is vital, as it resets the value of numberOfDots back to zero for the next iteration of the validation loop. Without this, the program would have a **logical error** because valid inputs would now be found to be bad by our validation algorithm e.g.



here the value 12 is found to be bad, even though it is certainly valid. This happened because I commented out the line above for demo purposes. Such a line of code is **very easy to forget and difficult to detect** when coding this sort of validation routine.

• After the validation loop finishes, the value is converted from a double to a float via **type-casting** as follows:

**radius = (float)radiusAsDouble;**

• the remainder of the method is the same as before

Take your time to fully understand the operation of the validation algorithm above as it is tricky. When trying to make sense of them, it should be useful to write down some **pseudocode** steps to outline the algorithm at a high-level.

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

Save the file **PerfectValidationForChoice.java** as **PerfectValidationForChoiceAndRadius.java**  in your Lab9 folder and make the necessary modifications to the **cylinderArea**() method.

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 with the following set of test values:

23, 45, 98, 123, 76.45, -889.3, thirteen, . , 34.5456f, abcde, %£”, -234, “” (return), 36000000000000000000000000000000000000000 (I think that’s about 40 zeroes!!)

**Re-Using the Validation Algorithm Above**

The routine we just looked at can be used directly in any situation where we want to validate a positive single-precision floating-point number. So, for example, if you had a program where you were getting the user to enter a sequence of positive numbers (whole or floating-point) and storing each in a **float** variable, then you could use the validation routine above. If the user was using a **double** variable instead, the validation routine would be exactly the same, except for the part where the conversion from string to double occurs i.e.

if(i==radiusAsString.length() && i!=0 && numberOfDots<=1 &&

!radiusAsString.equals("."))

{

radiusAsDouble = Double.parseDouble(radiusAsString);

if(radiusAsDouble>=-Float.MAX\_VALUE && radiusAsDouble <=

Float.MAX\_VALUE)

valid = true;

else

{

System.out.print("\n\nInvalid! Please re-enter the radius of the cylinder: ");

radiusAsString = input.nextLine();

}

}

The code above would now simply become:

if(i==radiusAsString.length() && i!=0 && numberOfDots<=1 &&

!radiusAsString.equals("."))

valid = true;

We will always take it that, as the double data type allows such massive numbers, both positive and negative, we will never enter a number that could contain that many zeroes! If we did, however, the program would be back in crash territory again. One makeshift way to prevent such a crash in this case would be to determine the number of characters in the input string before the decimal point (if there is one) and, if it exceeds 308 characters then disallow it. This is because valid double values in Java can have a maximum of 308 zeroes in them. You don’t need to worry about this possibility from the point of view of assessments or final exam though.

**Exercise 1**

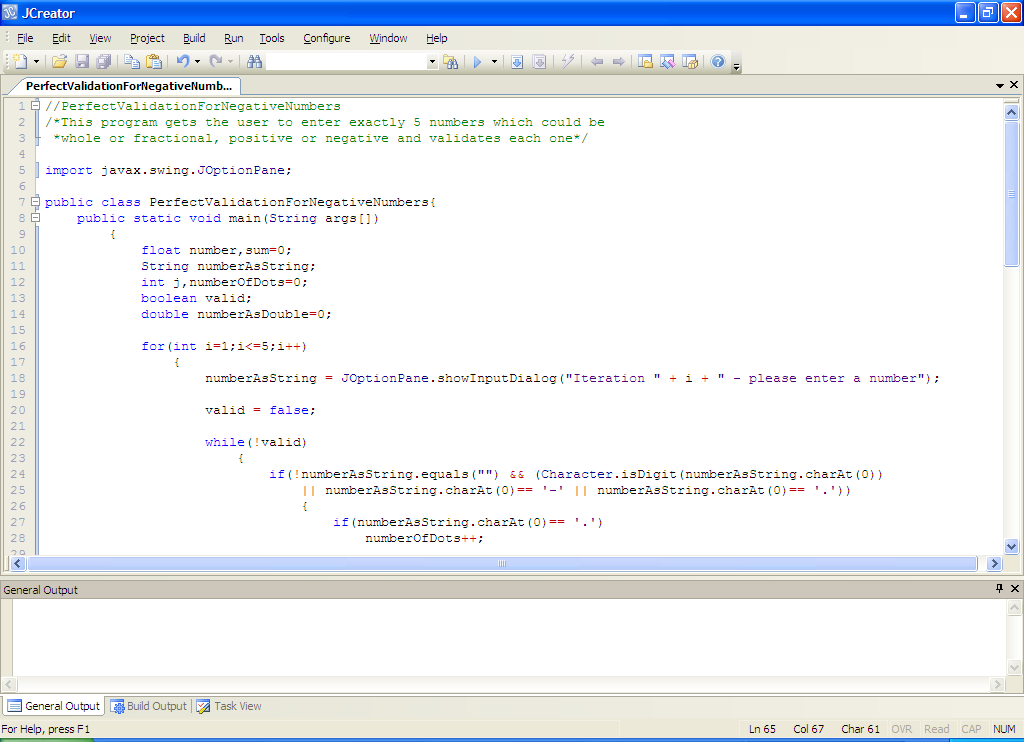
Save the **PerfectValidationForChoiceAndRadius.java**  as **Exercise1.java**. For practice now, see if you can code the validation routine for the **height** variable within the cylinderArea() method. Try to see how much of the validation routine you can remember.

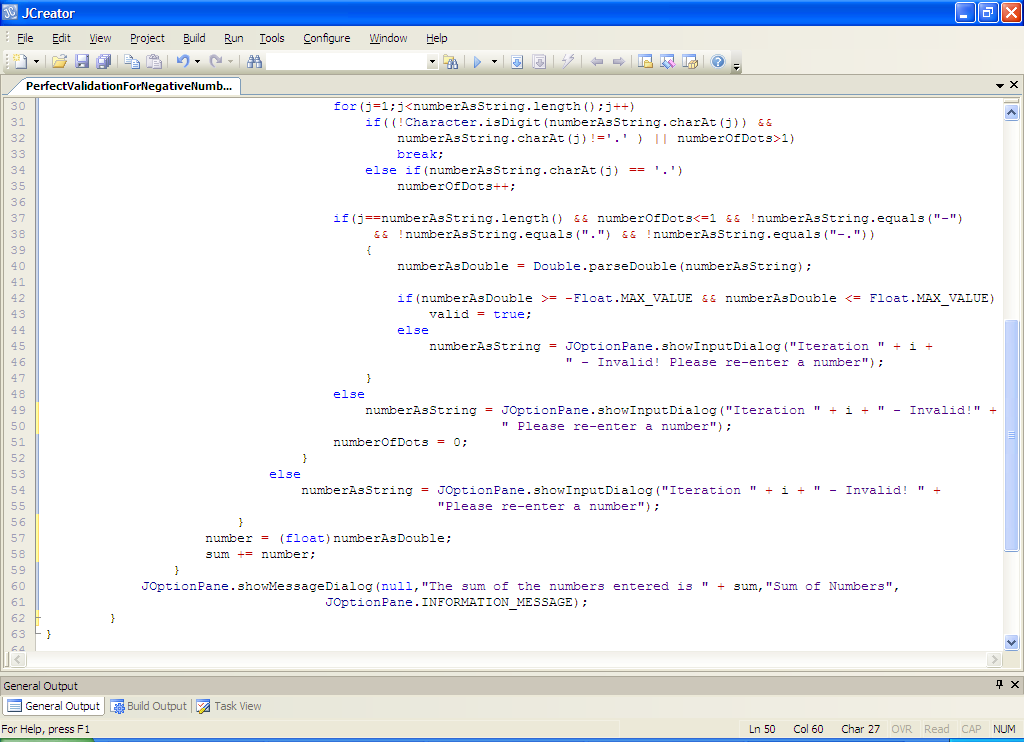
**Dealing with Negative Numbers**

In terms of validating numbers, our routines so far have always involved **positive** numbers, last time for the exam marks and this time for the radius of the cylinder. This was fine because these quantities are always expected to be positive numbers. However, these routines will break down for us if we wish to allow negative numbers into our programs, as they will be rejected. So, if we want to write a program that takes in and validates exactly 5 numbers, and these numbers can be positive or negative and can have a decimal point in them also, we will use a routine that is very similar to the one we used for the cylinder radius with minor modifications.

**Aim:** We wish to write a program that gets the user to enter exactly 5 numbers which could be whole or fractional, positive or negative and validates each one, before displaying the sum of the 5 numbers

**Java Code:**





**Analysis of Program:**

• The main loop in this program is a **for loop** which iterates 5 times. Each time it iterates the user is prompted for a number on an input dialog. The number is then validated to ensure it is a positive or negative whole or fractional number. Once the a valid number is supplied, the validation loop stops for that particular number and a running total of the numbers is kept as the main loop progresses. When the main for loop stops, the sum of the 5 numbers gets displayed on a message dialog.

• The validation loop begins with the valid variable set to false, to ensure we get into the validation loop in the first place. The first thing that happens within the validation loop is we check to see it the user actually entered something and, if they did, we check to see if the first character was either a digit, a dot or a – sign. If it fails any of these tests, we know that the input was bad so we issue an “Invalid!” message on the else section in line 53.

• Next there is a piece of logic that tests whether the first character in the users input was a dot. If it was, we must increment the dot count. Again, this is vital because a valid number can contain no more than 1 dot.

• the for loop that follows on from this is virtually a carbon-copy of the one used for validating the radius earlier. The only difference is we use a different name for the loop counter and the counter is set to begin at 1 rather than zero. This is because we have already analysed the first character of the input so we begin our analysis with the second character now.

• When this for loop has stopped, we must check to see whether or not it exited early. We also check to see whether the number of dots in the input was less than 2. There is also a check for a number of exceptional cases that must be handled to **prevent logical errors**. These cases are if the user enters “.” (as before), “-” or “-.” The reason these cases must be handled is that they all make it through the for loop without exiting early and all will satisfy the **j==numberAsString.length()** test within the if section. If the for loop exited early or the number of dots exceeded 1 or the user entered any of the special cases, then the else section on line 48 executes and the user gets the “Invalid!” message

• If everything is in order up to this point, the conversion from String to double takes place as before, with the comparison to **–Float.MAX\_VALUE** and **Float.MAX\_VALUE**. If the value lies in this range we have a valid float number so we set valid to true and the validation loop stops. If, however, the number entered was outside this range, the user gets the “Invalid!” message.

• As before, **numberOfDots** is reset to zero towards the end of the validation loop to **prevent a logical error**.

• Once the validation loop stops, the number is converted from double to float via **type-casting** and then added to variable sum to keep a running total of all the valid numbers. This is then displayed when the for loop completes.

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

Click the new file icon on the JCreator IDE and create a file called **PerfectValidationForNegativeNumbers.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 with the following set of test values:

23, 45, 98, 123, -, 76.45, -889.3, thirteen, . , 34.5456f, abcde, %£”, -234, “” (return), 500000000000000000000000000000000000000000 (40 zeroes here) , -.

**Exercise 2**

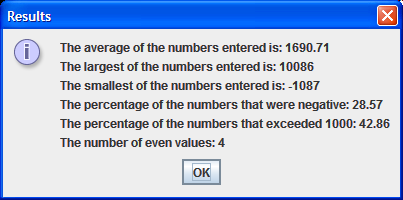
Write a Java program that reads in an arbitrary amount of positive or negative **whole numbers**, end of input is signaled by entering “x”. Each number entered should be perfectly validated. The program should determine

* The average of the numbers entered
* The largest of the numbers entered
* The smallest of the numbers entered
* The percentage of the numbers entered that were negative
* The percentage of the numbers entered that exceeded 1000
* The number of even values entered

The average and percentages should be displayed correct to **2 decimal places**. Use the **JOptionPane** class for input and output here.

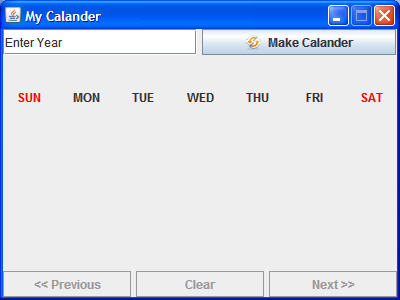
The following screenshot shows my results for the following set of valid inputs: -126, -1087, 10086, 111, 55, 1074, 1722. You can use these values also when testing out your own program.

I have omitted the input screenshots for brevity here but ensure that you **test your validation routine fully**, making sure it passes all tests, including text inputs, mixtures of text and digits, empty string and values above Integer.MAX\_VALUE and below Integer.MIN\_VALUE. Also test against odd-ball inputs like “-” and “-0” etc.



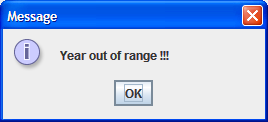
**Java in the Real-World - a little bit of R&R ☺**

What you’ve covered in this lab sheet and the last has not been easy, but it will hopefully help you to hone your problem-solving skills. Now we are ready to look at an example of Java being used in the real-world. The application we’re going to look at is a calendar application developed by an anonymous programmer from Sri Lanka and submitted to the code project <http://www.codeproject.com/Articles/14705/Java-Calender>. I have downloaded the code for you to have a look at and run. It is located in the **calendar** folder within the LabSheet9 folder on X: drive. It’s an interesting application and shows what you can do in Java with less than 300 lines of code.

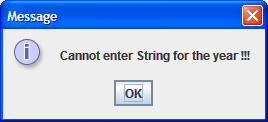


Spend a few minutes playing around with the interface now. Enter any year you like in the text-field and then press the “Make Calendar” button to see the calendar for January of that year. Then you can use the “<<Previous” and “Next>>” buttons to scroll through the months for that year.

We are covering validation at the moment, and for this calendar, only years between 1980 and 2099 are considered valid. If you enter a year outside this range you get



Likewise, if you try to enter a piece of text for the year, you get the message:



Once you are finished with the application, have a look at the code. You will see that there is a lot of GUI code in this application. We will be covering some of this around lab sheet 13 so you’ll be able to appreciate it better at that stage. But even now hopefully you can make out some of it intuitively.

The main number-crunching logic for the application takes place within the **setcal**() method. This contains a number of **for loops** which place the day numbers onto labels in the calendar GUI. It even has logic to deal with the possibility of a leap year to make an allowance for the extra day in February.

Take a few minutes to examine the code. See all the structures such as simple ifs, nested ifs and if-else for decision-making and see also how the application makes use of several user-defined methods such as makeGui(), clear(), setcal() etc. to break up the problem into a set of smaller, more manageable subtasks. In particular, see how short the main() actually is as a result of this modularization.