**Institute of Technology Tralee**

**Computing Department**

**Structured Programming 2**

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

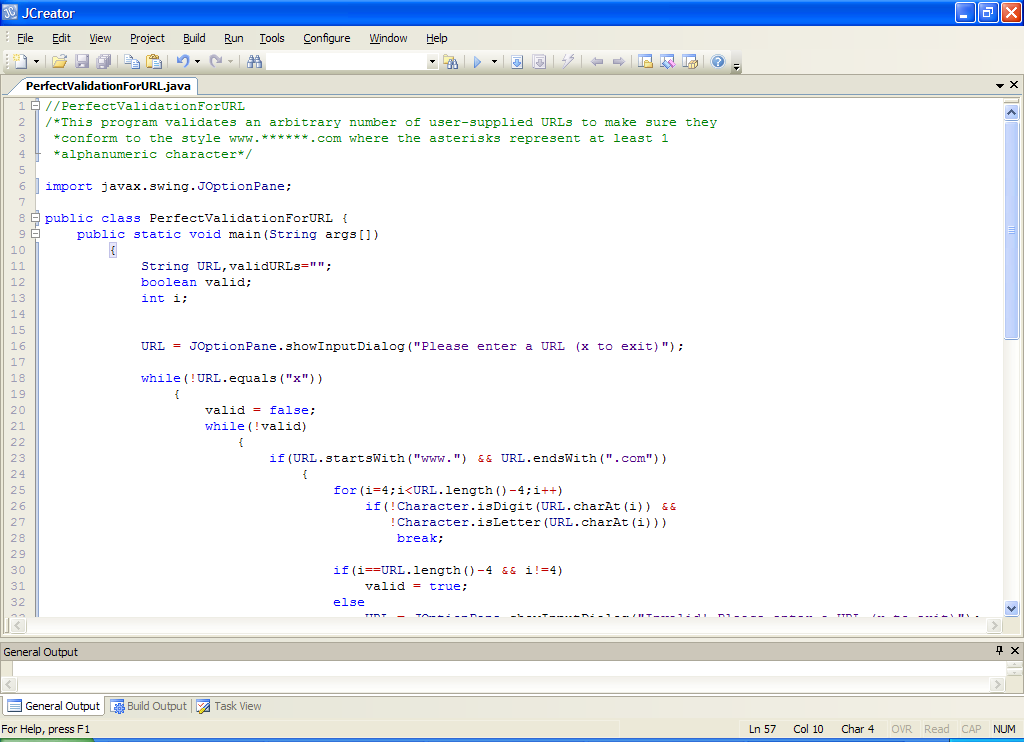
**Practical 10 – Input Validation**

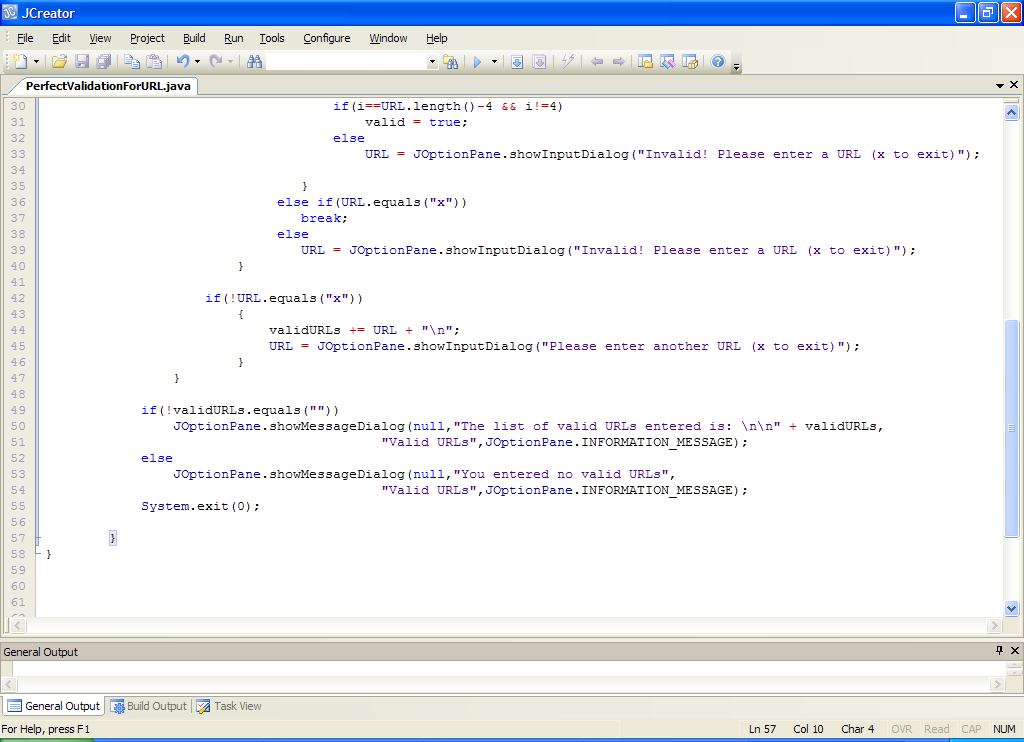
In this final lab sheet on the topic of input validation, we look at some further validation algorithms and we also look at creating **user-defined methods** specifically for the validation task.

**Validating a WWW URL**

**Aim:** We wish to write a program that reads in an arbitrary number of WWW URLs and validates each one. In our validation algorithm, we will take it that a valid URL has the form www.\*\*\*\*\*\*\*.com where the asterisks represent at least one alphanumeric character.

**Java Code:**





**Analysis of program:**

• The main loop in this program is a data-sentinel controlled while loop that keeps iterating until the user enters the sentinel value, which is chosen to be “x” here.

• Each time this loop iterates, the user-supplied URL is validated. You will recognize by now the use of the boolean variable valid, which gets us into the validation loop. The first part of the validation here checks to see whether the URL entered begins with the letters “www.” and ends with the letters “.com”, since these are requirements for any valid URL in our case. We use the String class pre-defined methods **startsWith**() and **endsWith**() to do the job quickly here.

If the URL turns out to be valid up until this point, we then proceed to do more processing, otherwise a test is performed to see whether the user happened to enter the sentinel value of “x”. If they did, the validation loop immediately stops through the **break** statement. If the user did not enter the sentinel value, they receive the “Invalid!” message and are asked to re-enter the URL.

• if the URL does begin with “www.” and ends with “.com” the next part of the validation code is:

**for(i=4;i<URL.length()-4;i++) if(!Character.isDigit(URL.charAt(i)) && !Character.isLetter(URL.charAt(i)))**

**break;**

It might look a little strange to see the for loop counter variable being initialised to 4 here, when you are so used to seeing it set to zero or 1. However, we are using this for loop to process the characters between the “www.” and the “.com” of the supplied URL. Since the first 4 characters have been validated, along with the last 4, we must skip these. Therefore we set i to 4 to begin with and we let it increase until we have reached the 4th last character in the URL, this is coded with **URL.length()-4**

Each time this for loop iterates, we just test to see whether the character in question was either a letter or a digit (alphanumeric). If it was then the for loop continues iterating, otherwise it stops prematurely through the **break**. We use the **isDigit**() and **isLetter**() methods here in our validation, along with **charAt**() for extracting each character.

• When this for loop stops, we then use the value of the loop counter i to determine whether the loop stopped early or not. If it stopped early, then there must have been something wrong with at least one of the characters enclosed between the “www.” and “.com”. The code:

**if(i==URL.length()-4 && i!=4)**

**valid = true;**

**else**

**URL = JOptionPane.showInputDialog("Invalid! Please enter a URL (x to exit)");**

firstly checks to see whether i matches the value of URL.length()-4

if it does, and the value of i isn’t 4, then the boolean valid gets set to true and the URL has been found to be valid. Otherwise, the user gets the “Invalid!” message and must re-enter the URL.

The possibility of i being 4 is a **special case** here that must be handled to prevent a **logical error**. It occurs if the user were to enter “www..com” as the URL. Hopefully you can make sense of this based on the if test above.

• Once the validation loop has finished, a check is performed to see whether the sentinel value was entered. If it was, then the main while loop exits immediately as the loop condition becomes false. If it wasn’t entered, then a valid URL must just have been entered, so we join it onto the variable validURLs whose job is simply to keep a list of all the valid URLs the user has entered. Then the user is asked to enter another URL.

• Once the main loop has finished, a message dialog displaying the list of valid URLs appears. If the user entered no valid URLs, they get a message indicating this.

**Organising your Work**

Within SP2Stuff, create a folder called **Lab10** 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 **PerfectValidationForURL.java** in your Lab10 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**. Try out the following set of input values here:

“www.”, “.com”, “wwg.123.com”, “www.123abc.cot”, “www..com”, “” (empty string), “www. .com”, “WWW.123.com”, “www.&\*(.com”, “www.ab$.com”, “www.123.com”,

“www.abc.com”, “www.microsoft.com”, “www.9to5.com”, “www.ABC.com”

**Exercise 1**

Write a Java program that reads in exactly 5 time values in the format **hh:mm** and validates them perfectly. The time must have exactly 5 characters as indicated, with the hours part going from 00 to 24 and the minutes from 00 to 59. Note that the validation routine can be written in a relatively short way here, in the form:

**while(*test expressions*)**

**time = JOptionPane.showInputDialog("Iteration " + i + " - Invalid! " +**

**"Please re-enter a time in the form hh:mm");**

However, you can use any algorithm you like, as long as it validates the times properly. Don’t forget the **empty string** case!

Once the time is validated, the program should give the user a message based on the time value as follows: if the time is between 00:00 (midnight) and 11:59, they should get a “Good Morning!” message, if the time is between 12:00 (midday) and 17:59, they should get a “Good Afternoon!” message and if the time is between 18:00 and 23:59, they should get a “Good Evening!” message.

To do this part, you need to extract the hours part from the time and then convert it to a number. You should find the **substring**() method from the String class useful for doing the extraction here.

An example of using it would be as follows:

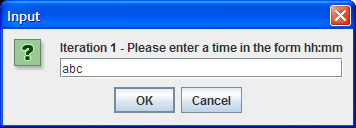
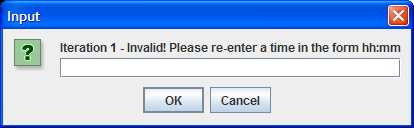
**String fruit = “apple”;**

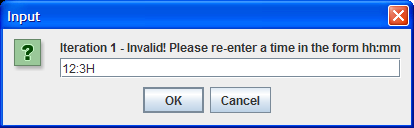
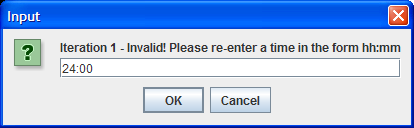
**System.out.println(“The first 3 letters of the word apple are ” + fruit.substring(0,3));**

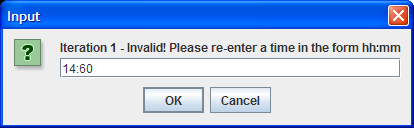
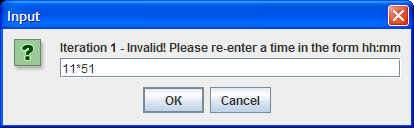
So it extracts the characters at position 0, 1 and 2 in this case i.e. “app”

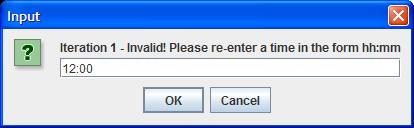
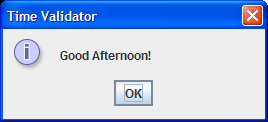
Note when using substring() that the first character goes by the number 0, as in the case for charAt(). Also note that substring() extracts characters up to, **but not including,** the position indicated by the second argument, which is often a source of confusion. So here, even though the second argument is 3, substring() only extracts characters up to and including position 2.

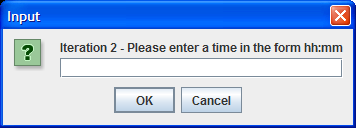
Your program will run as indicated in the following sample screenshots:

 and so it continues ……

**Exercise 2**

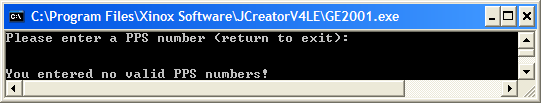
Write a program that reads in an arbitrary amount of PPS numbers and validates each one. A valid PPS number will have exactly 7 digits at the start followed by either 1 or 2 capital letters e.g. 1234567T would be a valid PPS number, as would 1234567AB. The program should keep a list of all the valid PPS numbers entered and display them when the user hits return. You can use the static predefined method **isUpperCase**() from the Character class here to check whether a character happens to be a capital letter. This is a boolean method which works along the same lines as isDigit() and the following code snippet shows it being used:

**if(Character.isUpperCase(character))**

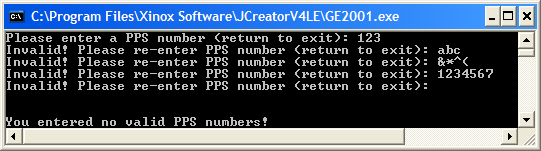
**System.out.println(“The character ” + character + “ is an uppercase letter”);**

The program will run as follows:

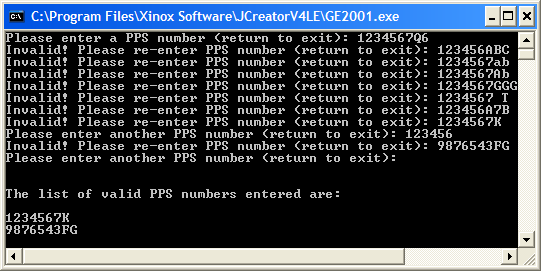
**Run 1: the user hits return immediately**



**Run 2: the user enters a few bad values before hitting return**



**Run 3: the user enters a mixture of bad values with a few good ones also**

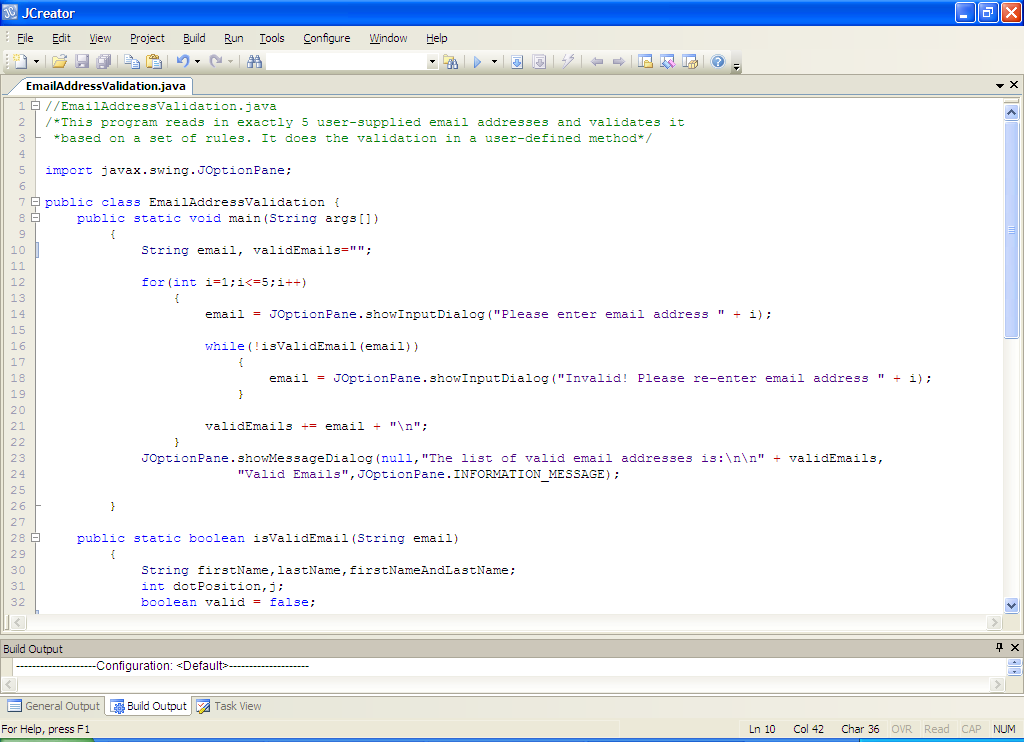


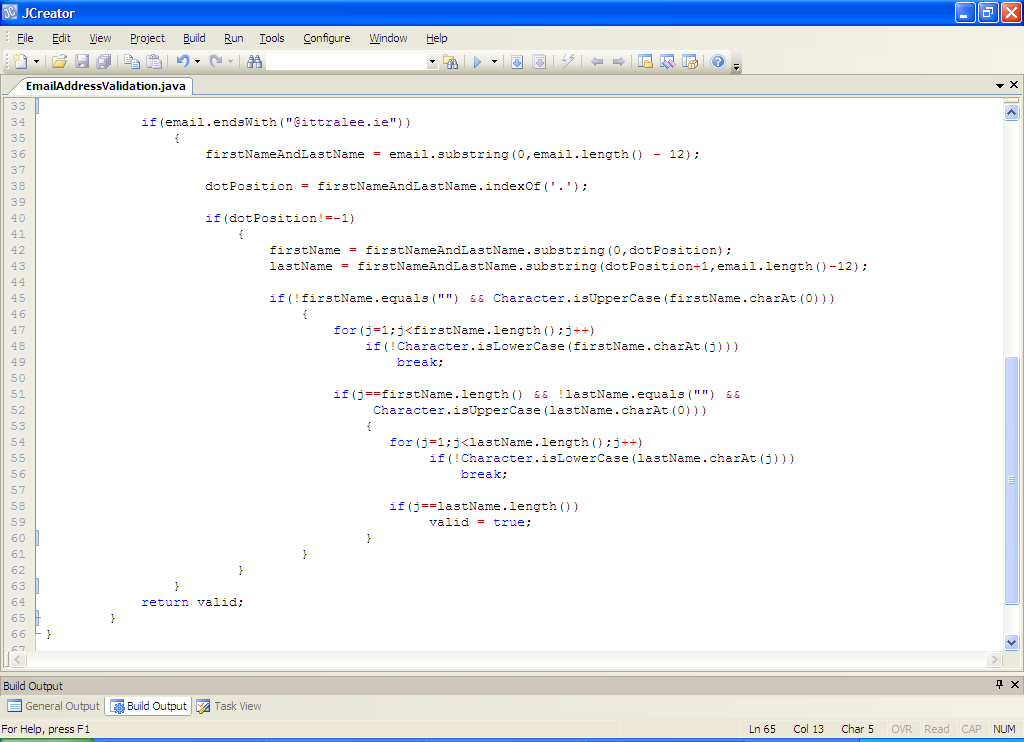
**Writing User-Defined Validation Methods**

Almost all the validation routines we have examined to date have taken place within the main(). There is nothing wrong with this, but you’ve probably noticed that it tends to make the main() quite lengthy. To rectify this, we will now look at putting validation code into separate methods and calling that method when needed.

**Aim:** We wish to write a program that reads in exactly 5 email addresses of the form Firstname.Lastname@ittralee.ie and validate each one. The Firstname part here must begin with a capital letter with all the rest lowercase letters. The same applies for the Lastname part. There must be a dot between the Firstname and Lastname parts and each must have at least one character. The validation code will be put into a user-defined method called isValidEmail()

**Java Code:**

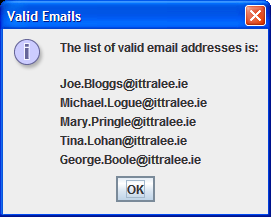




**Analysis of program:**

• Notice how short the main() is now. Only 3 variables are declared and it is much easier to read than when we put all the validation code within main(). So here we have a **for loop** that iterates 5 times and each time it loops, a new email address is read in. Each email is then validated through a call to the user-defined method **isValidEmail**() which takes as an argument the email value entered and validates it, returning the boolean value true if it is valid, and false otherwise. As long as the validation method keeps returning false, the validation loop keeps iterating.

When a valid email has been entered, the validation loop stops and the email is joined on to the variable validEmails, whose job is to store a list of all 5 valid emails. Once the for loop finishes, these valid emails are displayed, one per line as follows:



• As always in these programs, the difficult part is the validation. The method isValidEmail() begins by declaring all the variables that are needed.

• The first test is to check whether the email ends with the characters “@ittralee.ie” as required. The **endsWith**() method is used for this. Notice that if it passes this test, more testing will be done, but if it fails the test, execution jumps straight to line 64 where the code

**return valid;**

returns the outcome of the validation. In this case, because valid is initialized to false, the method will return a false value, and so the validation loop back in main() would issue the user an “Invalid!” message and ask them to re-enter.

• If the email should pass the first test, then the code

**firstNameAndLastName = email.substring(0,email.length() - 12);**

is used to extract the name part of the email address. It does this using **substring**(). We basically want to extract all the characters from the first one right up to the one just before the @ symbol. This is why the arguments to substring() are 0 and email.length()-12. The 12 comes from the fact that there are 12 characters in “@ittralee.ie”.

Next up is the code

**dotPosition = firstNameAndLastName.indexOf('.');**

which uses the String class’ **indexOf**() method, on the name part we have just extracted, to tell us the position of the dot character within the name (if it exists). We are doing this so that we can split up the name into a firstname part and a lastname part for further validation later.

• If the name part does not contain a dot character, then the call to **indexOf**() will return -1 to indicate this. So the next test is to check whether dotPosition is not -1. If it isn’t, we know there must have been a dot in the name part so we can do further tests. However, if dotPosition is -1, we know that the name part is invalid and so execution jumps straight to the

**return valid;**

code again, returning false from the method.

• If there was a dot in the extracted name, then the next thing we do is to extract the firstname and lastname parts individually with the code

**firstName = firstNameAndLastName.substring(0,dotPosition);**

**lastName = firstNameAndLastName.substring(dotPosition+1,email.length()-12);**

again, we use substring() to get the 2 parts of the name. You can see how dotPosition is being used to split up the 2 parts of the name. Hopefully you can make sense of the arguments to substring() in each case.

• Once the 2 parts of the names are extracted, the next part of the validation involves checking whether the first character of the firstname is a capital letter, which is a requirement for a valid email address here. The **isUpperCase**() method is used for this. Note here that, before we can extract the first character with charAt(), we need to firstly make sure that there is actually a character to extract, otherwise the program will **crash** at this point. This is the reason we check to see whether firstName holds the empty string. As long as the first character of the firstname is a capital letter, more validation will be done, otherwise execution jumps straight to the

**return valid;**

code again, returning false from the method.

• Once we have verified that the firstname begins with a capital letter, the code

**for(j=1;j<firstName.length();j++) if(!Character.isLowerCase(firstName.charAt(j)))**

**break;**

examines each of the remaining characters in the firstname, stopping as soon as a character is encountered that is not a lowercase letter. The **isLowerCase**() method of the Character class is used here to quickly determine whether a character is a lowercase letter.

When this loop stops, a test is then performed to compare the value of the loop counter variable j with the length of the firstname. We know that if j matches the value of firstName.length() that the for loop must have completed, and so all the remaining characters must have been lowercase letters as required. As part of this if structure, we also test to see whether the first character in the lastname is a capital. If it passes these tests, then we do our last piece of validation, otherwise execution jumps again to line 64 to return false.

• Should the firstname be valid, and the first character of the lastname be a capital letter then the code

**for(j=1;j<lastName.length();j++) if(!Character.isLowerCase(lastName.charAt(j)))**

**break;**

executes to check whether all the remaining characters in the lastname were lowercase letters. This algorithm is exactly the same as for the firstname earlier.

Again, if the for loop completes, then j will match lastName.length() and so all the remaining characters must have been lowercase letters, in which case our validation is now completed and we set valid to true, which is then returned from the method. Otherwise, at least one of the characters must have been a non-lowercase letter so execution jumps to line 64 to return false.

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

Click the **New File** icon on the JCreator IDE and save the file as **EmailAddressValidation.java** in your Lab10 folder. Now type in the code for the program above.

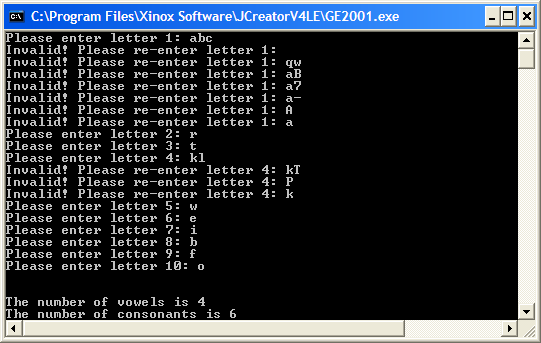
If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences 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:

“Joe.Bloggs@ittralee.it”, “Joe.Bloggs\*ittralee.ie”, “Joe.BloggS@ittralee.ie”, “JoE.Bloggs”@ittralee.ie”, “joe.Bloggs@ittralee.ie”, “Joe.bloggs@ittrale.ie”, “JoeBloggs@ittralee.ie”, “Joe.@ittralee.ie”, “.Bloggs@ittralee.ie”, “.@ittralee.ie”, “@ittralee.ie”, “J.Bloggs@ittralee.ie”, “Joe.B@ittralee.ie”, “J.B@ittralee.ie” “Joe.Bloggs@ittralee.ie”

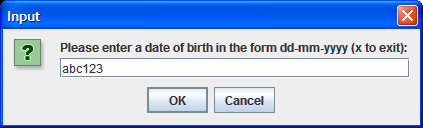
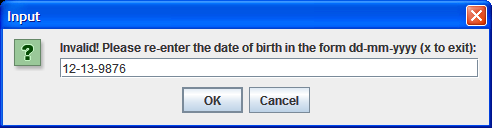
**Exercise 3**

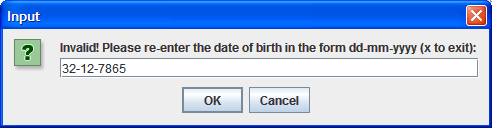
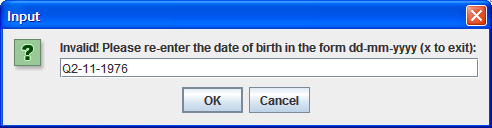
Write a Java program that reads in exactly 10 user-supplied lowercase letters and validates each one, using a call to a user-defined method **isValidLowercase**(). This method should take a String argument and return the boolean value true or false, depending on whether the validation resulted in a valid lowercase letter or not. Note that the validation method should be very short here. When all 10 letters have been entered, the program should display the total number of vowels and the total number of consonants. Your program will run as indicated in the following sample screenshot:

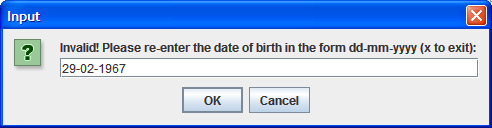
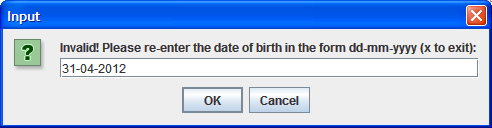


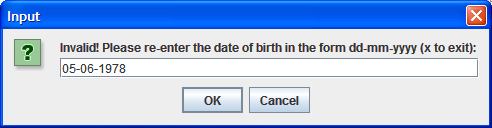
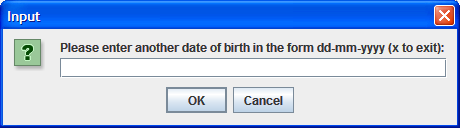
**Exercise 4**

Write a Java program that reads in an arbitrary number of date of birth values (x to exit). Each date of birth should have the form **dd-mm-yyyy** where dd is a number from 01-31 (depending on the month value), mm is a value from 01-12 and yyyy is any value from 0000 to 9999. The program does not have to handle leap years at all. You should validate each date of birth value through a call to a user-defined method **isValidDateOfBirth**() which takes a String argument and returns the boolean value true or false, depending on whether the validation resulted in a valid date of birth or not. The program should also keep track of any dates of birth that occurred in the 1960s and display these once the main loop has stopped. The program will run as indicated in the following sample screenshots:

…and following more rounds of input, finally the following dialog (your output here will just depend on the dates you actually typed in, make sure you type in a few in the 1960s):

