**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

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

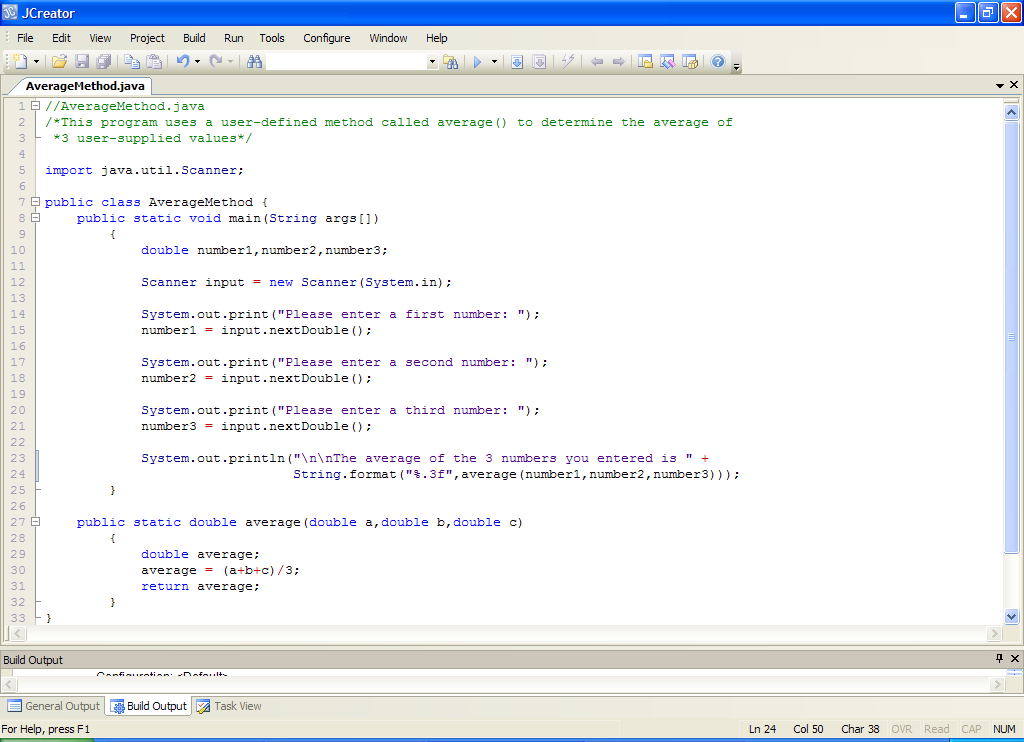
**Practical 5 – User-defined Methods**

In this lab we continue to explore methods. In particular, you will see how to write your own **user-defined methods** and then call them within a program.

**A First User-defined Method**

**Aim:** We wish to write a program that contains a user-defined method called average() which is able to determine and return the average of 3 user-supplied numbers.

**Java Code**:



**Analysis of program:**

• Here we create a console application. We use the Scanner class’s **nextDouble**() pre-defined method to read in the three user-supplied values from the keyboard.

• Look closely at the **println**() here:

**System.out.println("\n\nThe average of the 3 numbers you entered is " + String.format("%.3f",average(number1,number2,number3)));**

Notice that a call is being made to the user-defined method **average**() within the format(). This is not unusual though, remember that it is perfectly legal to make calls to methods from within other methods. We have done this many times in the past with pre-defined methods and it applies equally well for user-defined methods.

You can see that the average() method takes 3 arguments here.

• Notice that the main() ends after the println() but the **program code is not yet finished**. We now **define** our user-defined method **average**(). Without defining the method, the call above in main() would produce a **syntax error**.

All methods must be self-contained blocks of code – you **cannot define a method within another method** – a **syntax error** would result. Also, **methods must always be defined within the confines of a class definition** – to attempt to define one outside of a class would also result in a **syntax error**.

• The user-defined method is defined with the following header

**public static double average(double a,double b,double c)**

This header tells us a number of things

\* that the name of the method is average,

\* that the method expects 3 arguments to be passed to it when it is called

\* that the method expects each argument to be of type double

\* that the method returns a double value at the point it was called

\* that the method is static

\* that the method is public

It is often overlooked, but when designing a method, **one of the key considerations is the name** of the method. It **should be meaningful** and describe concisely what the method actually does. Since this method calculates the average of 3 numbers, the name average is reasonable.

In this case the method aims to calculate the average of 3 numbers, hence there will be three arguments. Note that the arguments are called a, b and c here. Of course, you can call these whatever you like as long as they are legal variable names, but they too should be meaningful.

In order for the method to be able to calculate the average of *any* valid Java numbers, the data type **double** is used because it is effectively a superset of float and int. This means int and float values can be passed into the method and they can still be used as valid arguments.

As the method aims to determine and **return** the average of the 3 numbers it receives as arguments, it must therefore have a non-void return type. In this case, it makes sense that the return type is double since the average of 3 double values is also a double value.

Do not get too hung up on the **static** keyword here – just remember for now that **every user-defined method you will create at the moment will have to be static**. This is because the method is being defined within the same class as main() which is itself static and there is a rule in Java stating that a static method (such as main()) can only make calls to other static methods defined in the same class as itself.

Don’t worry about the **public** keyword here either – in fact, this method does not have to be public, but I have chosen this keyword for the method since the vast majority of methods (user-defined and pre-defined) are public. Out of convention, you should always make your user-defined methods public for now.

• **All method definitions begin with an opening curly brace and end with a closing curly brace**.

The **general form** of a method definition is as follows:

*returnType methodName* ( *arg1Type arg1, arg2Type arg2,.......,argnType argn*)

{

//statements making up the method body

}

• The code within the curly braces is the **method** **body**. Here the body contains 3 statements.

The first declares a variable called average. This will store the average that has been calculated.

The second statement performs the average calculation. It uses the arguments a, b and c and just sums them and divides by 3 for a simple average. This is then stored in average.

Finally the third statement returns the value stored in the variable average using a **return** statement.

return average;

Note that this statement is vital since, without it, there is no way to pass the value calculated by the method to another location in the program. As you might expect, return is a **keyword** in Java and as soon as a return statement executes, **control is passed back to the point in the program where the method was called**. For this reason, a return statement is the very **last thing that happens** within the body of a method. If you do put statements after the return then the compiler generates a syntax error as the statements are “**unreachable**”.

**Organising your Work**

You should have a folder under X: called OOP1Stuff. This time, create a folder called **Lab5** within it 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 **AverageMethod.java** in your Lab5 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. With proper testing you should see that this program is **not validated**. We will cover validation in future lab sheets.

**Improving the average() method**

The **average**() method above contains 3 lines of code to do the job. A lot of students (and pros alike) would code the method in such a way and the important thing is that it works correctly. However there is a **shortcut** to coding such a method as follows:

**public static double average(double a,double b,double c)**

**{**

**return (a+b+c)/3;**

**}**

Now we have reduced the lines of code from 3 down to 1 and saved ourselves the expense of the average variable altogether. Keep it in mind when you are coding methods that involve returning values generated from formulas. You should now prove to yourself that the above code works okay by **modifying** your original program and recompiling/running.

**Calling Methods**

When calling methods, it is vital that you **call them correctly**, in accordance with how they were defined. In order to call a method correctly, you need to look at its **method definition header**. You will get this information from the Java API if the method was obtained from there. Otherwise, the method is probably your own anyhow and the method definition header can easily be examined.

For example, imagine we wish to use the **pow**() method from the Math class to do a quick calculation. From the API we see its method definition header is:

|  |  |
| --- | --- |
| static double | [**pow**](http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html#pow%28double,%20double%29)(double a, double b) |

Therefore, the following method call:

System.out.println("The value of 2 to the power of 3 is " + 2\*Math.pow(3));

is **invalid** because there is no such method called pow() that takes just one argument defined in the Java API Math class – it requires two arguments. In this case the **number of arguments is incorrect** and we get a **syntax error**.

Equally the method call

System.out.println("The value of 2 to the power of 3 is " + Math.pow(3,2));

is invalid because, even though the number of arguments is correct, the **order is incorrect** and we will end up with a **logical error** since we will be calculating 3 to the power of 2 rather than 2 to the power of 3.

Likewise the method call

System.out.println("The value of 2 to the power of 3 is " + Math.pow(“two”,“three”));

is invalid because this time we are attempting to pass in String constants to the method, but the method expects arguments of type double. Now the **type of the arguments are incorrect** and we get a **syntax error**.

And finally the call

System.out.println("The value of 2 to the power of 3 is " + pow(2,3));

Is invalid because the pow() method is **static** and as such must be called on the class in which it is defined, in this case the Math class. So again, we get a syntax error.

The call should have been:

System.out.println("The value of 2 to the power of 3 is " + **Math.pow(2,3)**);

When it comes to calling methods, you need to know 6 critical things

● The **name** of the method (the exact spelling and case since method names are case-sensitive)

● The **number of arguments** that the method expects

● The **type of arguments** the method expects

● The **order of the arguments** the method expects

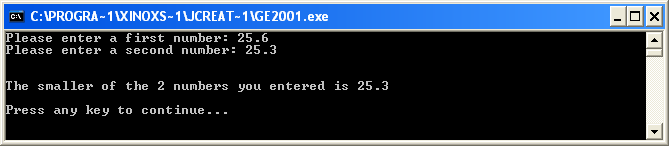
● The **return type** of the method

● Whether or not the method is **static**

If you get any of these wrong, you will most likely end up with a syntax error.

**Exercise 1**

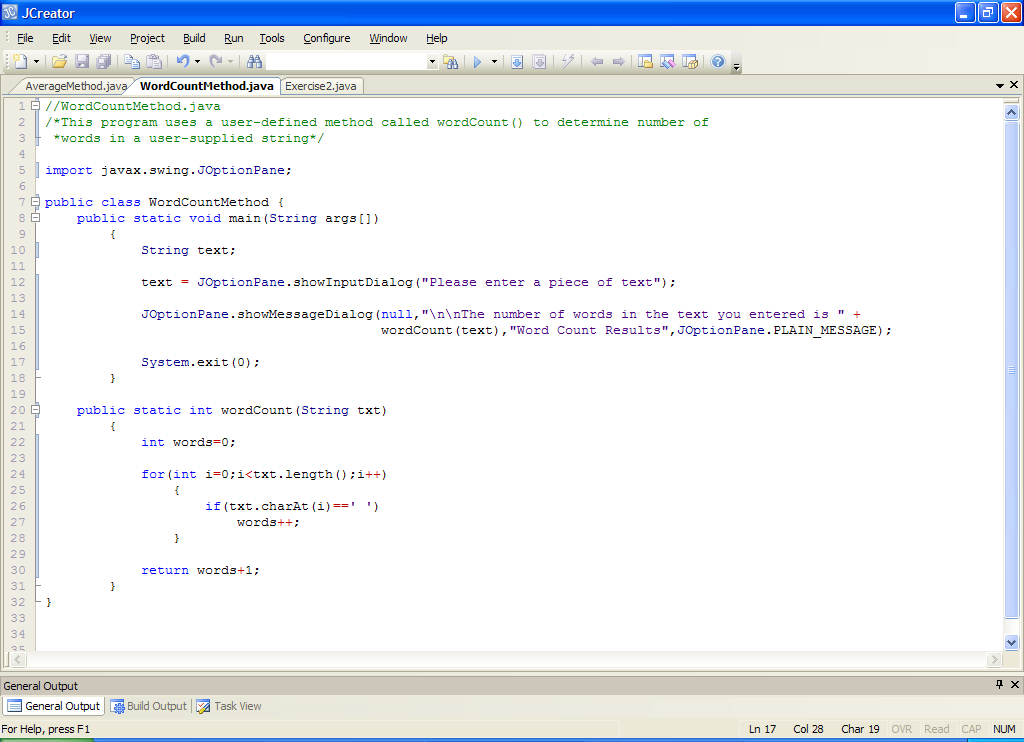
Write a program called **Exercise1.java** which uses a user-defined method called **minimum**() that takes 2 user-supplied float values as arguments and returns the smaller of the 2 to main(). The main() method should then output the smaller value. If the numbers turn out to be equal, it just returns either number. You can write the program as a console application and it should run as indicated in the following screenshot:



**Another Example of a User-defined Method**

**Aim:** We wish to write a program that contains a user-defined method called wordCount() which is able to determine and return the number of words within a piece of user-supplied text.

**Java Code**:



**Analysis of program:**

• The user is prompted for a piece of text which is read in using an input dialog.

• Within the message dialog, there is a call to the user-defined method wordCount(). The method is passed the text the user just entered as an argument.

• After the main() method, we have the method definition header for wordCount()

**public static int wordCount(String txt)**

we can tell from this

\* the method name is wordCount

\* the method takes a single argument

\* the type of argument the method takes is a String

\* the method returns an integer value at the point it is called

\* the method is static

\* the method is public

• The method body for wordCount() contains a variable declaration. Here the variable words is initialized to zero, as it is a counter variable. We use this to keep track of the number of words in the text passed in as an argument.

• next up is a **for** **loop** which iterates a total of **txt.length()** times. The number of times it iterates depends on the text entered by the user and how many characters it contains.

• each time the for loop iterates, a character is extracted from the text using **charAt**() and a simple if test is performed to check whether the character is a space. If it is, we say that this constitutes a new word and so the counter variable words gets incremented.

• When the loop finishes, there is a **return** statement which returns the value of the variable words, with 1 added to it to compensate for the fact that the last word will have no space following it.

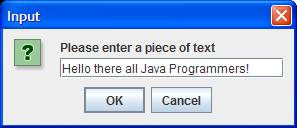
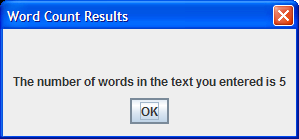
So you can see that pretty much anything that can be done within main() can also be done within a user-defined method e.g. decision-making, looping etc.

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

Click the **New File** icon on the JCreator IDE and save the file as **WordCountMethod.java** in your Lab5 folder. Now type in the code for the program above.

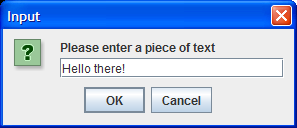
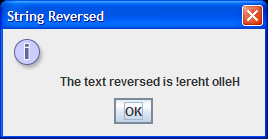
If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences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

**Exercise 2**

Write a program called Exercise2.java which uses a user-defined method called reverse() which takes a single String argument and returns it in reverse order. The main() method should then output the reversed string. Note that you’ll need **charAt**() for this method. Your program should run as indicated in the following sample screenshots:

**boolean Methods**

There are many occasions where there will be a need to write a method that performs a test and the outcome of that test will need to be returned from the method. These methods use the **boolean** data type as their return type. We have not used this before in our programs but you will see it quite a bit from now on.

The boolean data type is very simple in that it only supports two possible values – **true** and **false**. Indeed, when I was discussing decision-making and looping structures, I often referred to the fact that the test expression associated with the structure would evaluate to either true or false, but I never mentioned the boolean data type at the time. However, all such test expressions must evaluate to a boolean outcome of true or false in order to be valid. For instance, if I had the code:

**if(age = 18)**

**System.out.println(“You are 18 years old”);**

There would be a **syntax error** because, although it is legal to assign the value 18 to the variable age, the type of the variable age is an int and so the test expression evaluates to an int rather than a boolean, which is illegal. The error here is that the programmer has forgotten to use **==** rather than = for comparing the variable to 18. This is quite common.

boolean methods are ones that have a return type of boolean. These methods typically perform a test and then return an outcome of true or false when called.

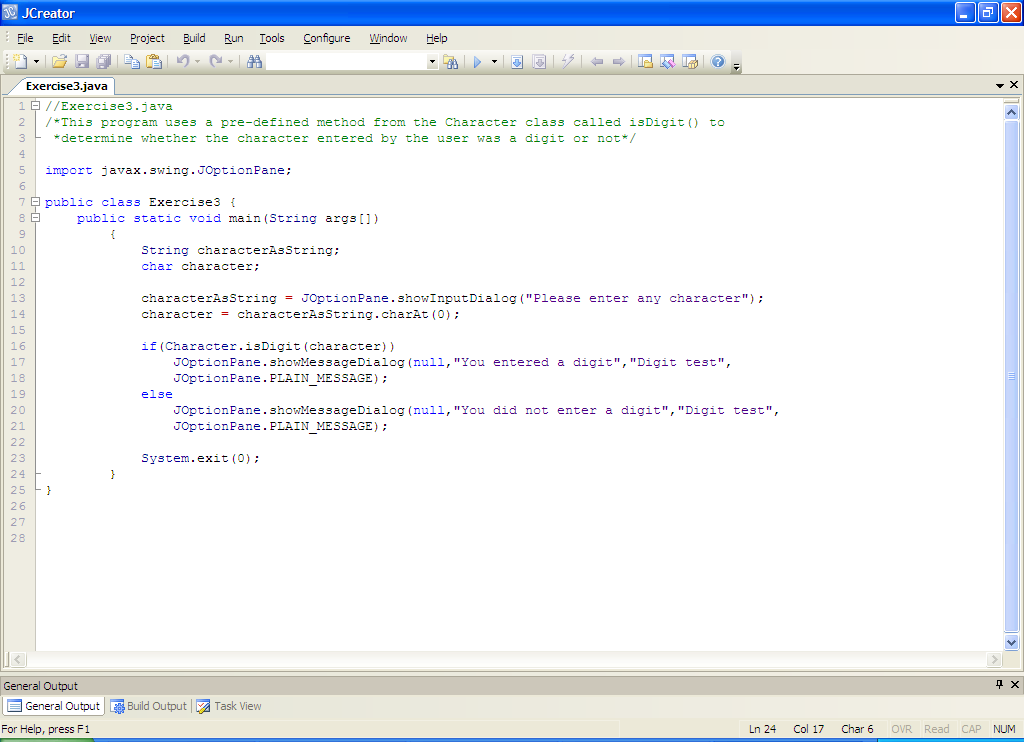
You have met an example of a boolean method already with the predefined **isDigit**() method of the **Character** class. This method has the header:

|  |  |
| --- | --- |
| static boolean | [**isDigit**](http://docs.oracle.com/javase/7/docs/api/java/lang/Character.html#isDigit%28char%29)(char ch) |

So we see that it is a method that expects a single char value to be passed to it when it is called, and then it returns a boolean value of true or false

You will see that boolean methods within the Java API often begin with the word “**is**”.

I got you to attempt to use this method in the last practical sheet to determine whether a user-supplied character was a digit or not and to display the outcome in main(). The main() in this case looks as follows:

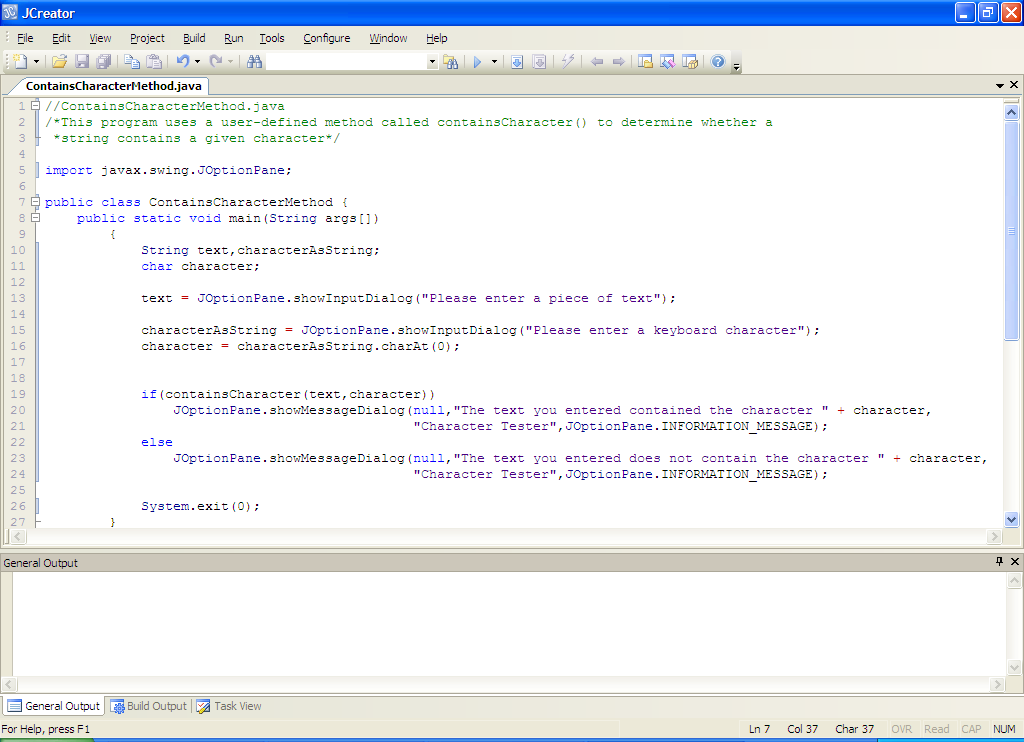


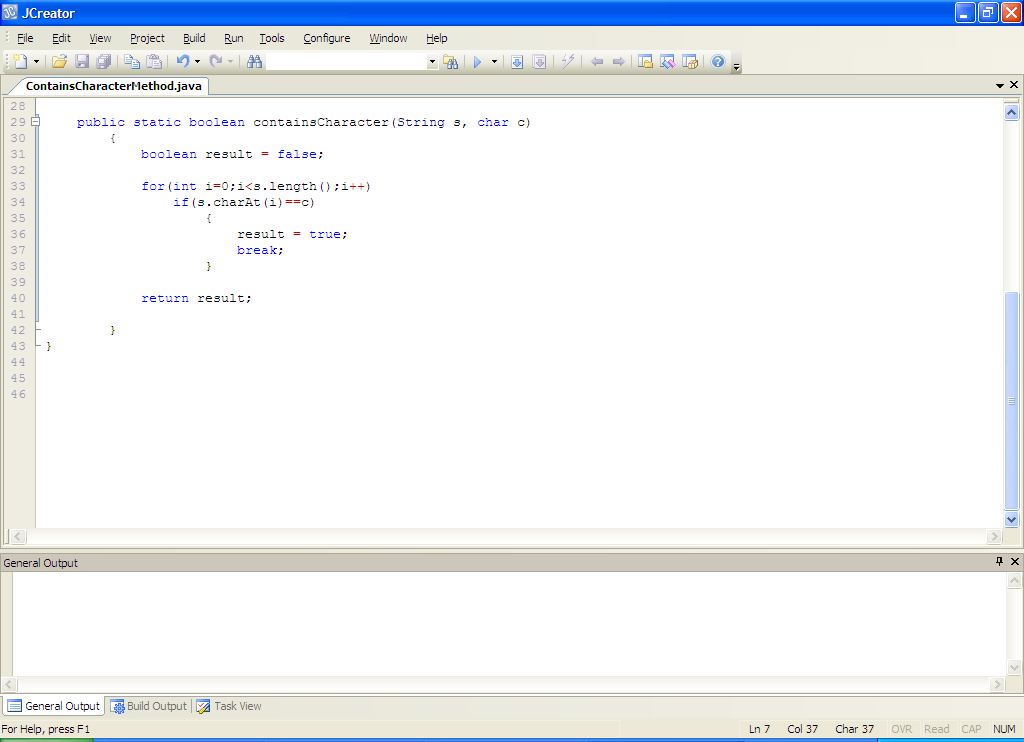
The boolean method isDigit() gets called and the value entered by the user (stored in variable character), is passed to the method at this point. The method call itself, because it will return a boolean result, forms the test expression for the if-else structure here. So, if the user did enter a digit character, the call to isDigit() will return **true** and so you get the first message. Otherwise, the call to isDigit() returns **false** and the else section is executed.

**A User-defined boolean Method**

**Aim:** We wish to write a program that contains a user-defined method called containsCharacter() which is able to determine whether a user-supplied piece of text contains a certain character.

**Java Code**:





**Analysis of program:**

• The user is prompted for a piece of text and a keyboard character, which are read in using input dialogs.

• The character read in is converted from a string to a char using **charAt**().

• Next there is an **if-else** structure that makes a call to the user-defined method **containsCharacter**(). The text and character entered by the user are passed to the method as arguments at this point. Depending on the outcome of the method call, which will return a **boolean** value of true or false, either the if or the else section will execute to give the appropriate output.

• After the main() is the method definition header for containsCharacter()

**public static boolean containsCharacter(String s, char c)**

The method takes 2 arguments here and returns a boolean. As usual, it is public and static.

• The method begins by initializing a boolean variable called result to false. The purpose of this variable will be to store the result of the test we will perform within the method i.e. to see whether the string actually contains the character or not. The variable **must be a boolean** also as we are going to return the value of this variable when the loop has completed its work and it must match the return type specified for the method. We initialize the value of result to false to begin with because at this point we can only assume that the character does not exist within the string (as we haven’t done any testing yet).

• Next there is a for loop that potentially iterates a total of **s.length()** times. It could exit early however, because there is a **break** statement within its body. We want the possibility of an early exit here because we may find the character very early on and then there is no point in checking other characters at that point.

• The **simple if** test within the for loop does the testing to see whether the character entered matches a character extracted from the string. If it does, then the variable result gets set to true and the loop immediately stops. Otherwise the loop keeps going for another iteration, potentially up until the last character in the string has been extracted.

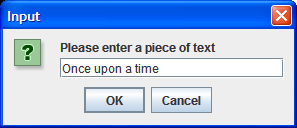
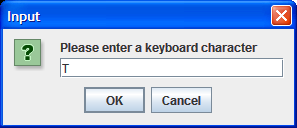
• Once the loop stops, the value stored in variable result gets returned.

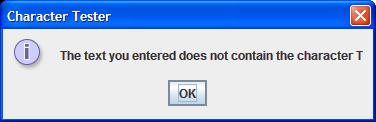
**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **ContainsCharacterMethod.java** in your Lab5 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. A sample run is as follows:

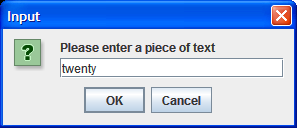
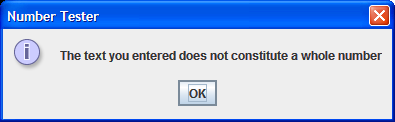
 



**Exercise 3**

Write a program called **Exercise3.java** which uses a user-defined method called isWholeNumber() that takes a single String argument and determines whether or not it constitutes a whole number (positive or negative). If it is a whole number the method should return the boolean value true, and return false otherwise. Use the **isDigit**() method from the Character class in your coding here. The main() method should then display the result. You can ignore the possibility that the user enters the empty string here. Your program should run as indicated in the following sample screenshots:

Run 1:

Run 2:

