**Institute of Technology Tralee**

**Computing Department**

**Introduction to Programming**

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

**Practical 1 – Programming Basics**

Welcome to the “Introduction to Programming” module! This introductory module aims to teach the basics of a **structured, high-level programming language** and to introduce a disciplined approach to **problem solving**, **algorithm development** and **testing** software. The main thrust of the **module centers around** **you** - you will obtain **hands-on experience** of designing solutions and then writing, debugging and testing code using the programming language **Java,** while I will guide you along the way.

**Module Syllabus Content**

* Keyboard input and screen output
* Comments, variables, constants, operators and expressions
* Simple calculations and conversions
* Selection control structures
* Repetition control structures
* Problem-solving and algorithm development

**Exams**

It is expected that you will be given **2 continuous assessments** during this module. The first will be worth 10% and the second will be worth 20%. These assessments will be practical in nature where you will have the computer and Java available to you and **all of the marks will go for your coding ability**.

The **final exam** in December/January will be **worth 70%** and note that this will be a **written examination** testing your knowledge of the entire module, so you **must also** **get used to writing code on paper**.

**What do You Get from Successful Completion of this Module?**

* The ability to design, code, debug and test relatively simple Java programs.
* The ability to translate thoughts on paper into actual "real-life" Java programs.
* Familiarisation with the basics of a structured high-level programming language which can be used to learn other programming languages much more easily.
* Recognition of, and the ability to use, some of the most fundamental concepts in object-based programming such as classes, objects and methods.

**What is a Computer Program?**

A computer program is a **set of instructions** written in some programming language that is intended to accomplish some task.

All computer programs in this course will be written in the **Java** programming language.

**Introducing Java**

Java is a **structured**, **high-level**, **object-oriented** programming language. It is relatively new, first released in May 1995. It was created by **Sun Microsystems** in the USA.

**What is a Structured Programming Language?**

Java is a **structured** **programming language** in that programs written in it are **structured** in a particular way, mainly through the use of featuressuch as **selection control structures** and **repetition control structures (**which you will spend a lot of time learning about in this course).Other examples of structured programming languages are **PASCAL**, **FORTRAN**, **COBOL**, **C and C++.**

**What is a High-Level Programming Language?**

All of the programming languages mentioned above, including Java, are considered **high-level** **languages** because the **syntax** of the languages is **relatively straightforward to understand and interpret** and is **relatively close to English** in many ways.

There are programming languages that are **low-level** also, such as **assembly language** and **machine code**, which are much more difficult to understand but (thankfully) very few programmers require knowledge of these specialist languages. All the programming you will do at the ITT will involve high-level structured programming languages.

**Why use Java?**

Java was chosen to teach this module primarily because it is currently a **very popular language in the software development industry**.

Important also is the fact that the language is **completely free**, available for download from <http://www.oracle.com/technetwork/java/javase/downloads/index.html> .

Also it is a very **flexible language** meaning that **many different types of application** can be written using Java such as games, business applications, educational applications, multimedia applications, mobile applications, networking applications etc.

Java is a **highly portable** language also, meaning that Java applications written on one operating system, such as MS Windows can be run, without modification, on a different operating system, such as a Macintosh or Linux, **as long** **as there is a Java interpreter installed on the operating system**.

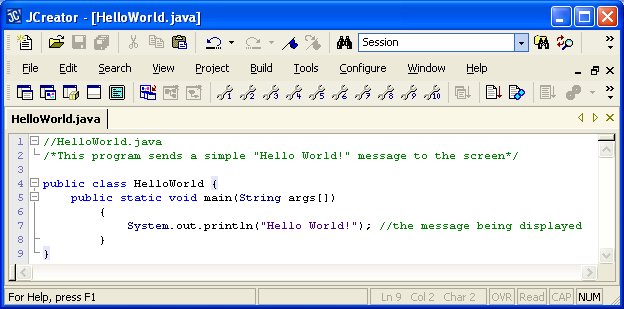
There is a **massive amount of support** for the language also with many forums and tutorials available online. One forum is at <http://www.javaranch.com> where you can ask any question you like on the language. Also, on many sites, users post their own code for games etc. so there is no shortage of useful information available.

The **language is OO (Object-Oriented)** also, bringing with it all the advantages of such a language, primarily **software reuse**. We will discuss the OO nature of Java further in the **OOP1 module** after Christmas.

**Basic Structure of a Java program - The "Hello World!" Example**

**Aim**: The purpose of this program is to output the message "Hello World!" to the monitor.

**Java Code**:



**Program Analysis**:

• At the top of the program is a **single-line** **comment**.In Java, a single-line comment begins with **two forward-slashes**.

• Comments are **special** in Java (and programming languages in general) because their contents are completely **ignored by the compiler (which analyses the program for syntax errors)**. They are a **style feature** which are used mainly to **explain something within** a program’s code or to **state the overall purpose of a program**. Here I have used it to state the file name of the program. They do not affect the running of a program in any way whatsoever and are **not output** to the screen themselves. You can include a comment **almost** **anywhere** in a Java program. It is considered **good programming practice** to always comment your programs appropriately so that someone reading your code can figure out what is going on more easily.

• The second line of code is a **multi-line comment**. These comments can spread out over multiple lines as their name suggests. They begin with the characters /\* and end with the characters \*/ You can put virtually any text you like in between.

• Comments are **meant to help the readability** of your programs and my own **style** is to always put a single-line comment at the top of my Java program which indicates the program’s file name, followed by a multi-line comment that summarises the purpose of the program. Many professionals adopt a style similar to this also.

• As mentioned earlier, **comments can appear almost anywhere within a program** and can be added at the end of a line of code also. I have another single-line comment added at the end of line 7 above to explain what the line of code is doing.

• The next line of code is

public class HelloWorld {

This code **defines a class** called HelloWorld. Recall that Java is an OO language and one of the things that differentiates such languages from procedural languages like C is the fact that structures such as classes exist. **Every program you write in this module will contain a class definition** because every Java program must have at least one class defined in it for the program to be of use.

It is into the class definition that we will be adding our code. For now, it may be useful to think of the class definition as simply a container into which we add our code.

The beginning of every class definition is marked by the character { and the end of every class definition is marked by the character } (this is on line 9 above).

The word class is a **keyword** in the Java language. All programming languages contain keywords, which are words that have a special significance in the language. Java has about 50 keywords in total and you will see many of them throughout this module.

After the class keyword, the **name of the class** always follows – in this case it is called “HelloWorld”. By convention, the **names of classes in Java begin with an uppercase letter and the first letter in every word after that is also in uppercase**. It is good programming practice to **adhere to these conventions** so that your programs look professional and will be more readable also.

Notice that the word **public** is used here – we are creating a public class called “HelloWorld”. For now, we will not go into the full meaning of this keyword. However, one syntax rule in Java is that, when you have a public class in a Java program, the **name of the program (when you go to save it) must match the name of the public class**, otherwise you get a syntax error.

• The next line of code is

public static void main(String args[])

This line of code defines a **method** called main(). A method is a self-contained block of code that is used to carry out a particular task. You can always recognise methods in Java because there are always **parentheses** attached to them.

The main() method in Java is a **very special** one though. It **marks the starting point of every Java application** when it runs. Therefore, **every Java application needs to have a main() method** **so that it can actually run**. Without it, the program can actually compile – because not every individual Java class needs to have a main() - but it cannot execute. The reason a program cannot run without a main() is because the Java runtime system looks to main() in order to locate the first instruction to be executed.

Note also that the main() method must be defined exactly as indicated above. It must be **public**, it must be **static,** it must have a **void** return type and it must take a **String** **array** (String [] args) as an argument. Any deviation from this will either mean a syntax or a runtime error. Also note that the main() method **must have an opening { and closing } to mark its beginning and end**.

**static** and **void** are also Java **keywords** and **String** is the name of a *predefined* Java class (predefined just means the class already exists). You might have guessed that String was a class because it **begins with a capital letter**.

Don’t worry about any of these details for now – just remember to put in these things when writing the main(). It might look terrible right now but it will become second nature in no time. The focus at the moment is **just to get familiar** with the “look” of a Java program.

• The line of code

System.out.println("Hello World!");

displays the text “Hello World!” to the output window, followed by a new line.

Here **System** is another example of a Java predefined **class** (how can you tell this?). **out** is referred to as the **output object**, which by default is the **output window** that appears when the program runs.

**println()** is an example of a **predefined method** that is being called here on the System.out object – this has the effect of displaying the text “Hello World!” to the output window when the program runs. When it has displayed its text, it brings the cursor onto the start of the following line (that’s what the “ln” at the end of “println” refers to)

In Java, all text messages must be enclosed within **double-quotes**.

Note that this line of code ends in a **semi-colon**. This is used to **mark the end of a particular statement** in the program. All Java statements will be terminated with one. Note that there are no semi-colons at the end of the lines that create the class or method definitions though, this is because **these are structures** rather than statements.

**Getting Started on the PC**

Switch on your machine if it is turned off.

The machine will boot to Windows 7 by default. Press ALT+CTRL+DEL for the login box. If available, type in your student t-number and password when prompted. Click **OK** to log on.

You should now be logged in to the Windows 7 operating system and the college network.

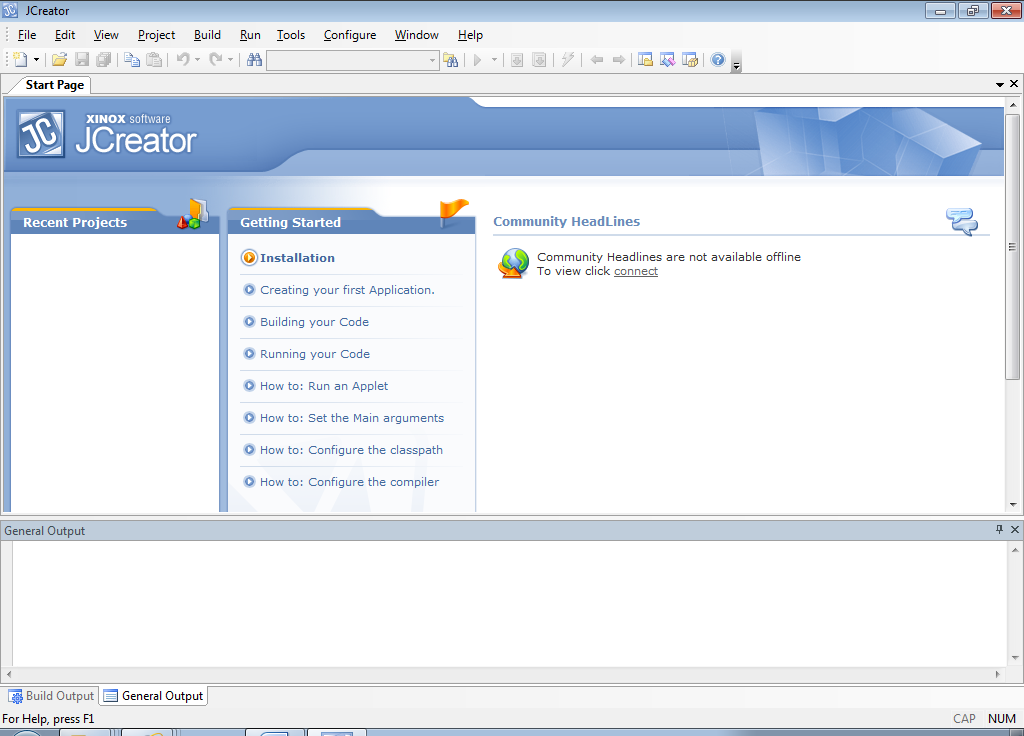
**Getting into JCreator LE**

Click on the **Start** button (bottom left of screen)

Go to the following location on the system:

**All Programs**->**Java**->**JCreator LE**->**JCreator LE 4.50**

The JCreator **Integrated Development Environment** (**IDE**) should now launch and you are ready for coding!



There are various menu options available along the top including **File**, **Edit**, **View, Build etc.** Each of these options gives a **submenu** with its own list of choices.

Use the mouse to choose the **File** menu option now - click the mouse on the word “**File**”. You are then presented with the submenu. Click the mouse on whatever option is required. As is normal for Windows applications, the **File** menu allows you to do such things as create a new file, open an existing file, save a file etc.

Have a look at the **Edit** menu now. You will see the normal options available in the sub-menu such as cut, copy and paste as well as others. This will be another handy menu for you to use as you program.

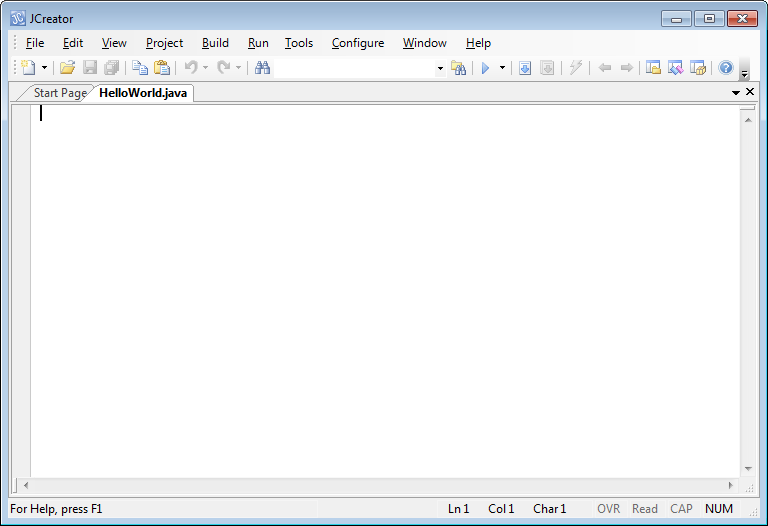
**Setting up your Folder Structure**

Create a folder called **JavaStuff** on your X: drive and within this folder create a folder called **Lab1**. This is where you will save all your work from this lab.

**Typing in your First Java Program**

To open up the **code editor window**, into which you will be typing your code, choose **File** from the main menu and then **New** and then **File**. Better again, you can just click the **New** **file** **icon** immediately below the File menu.

A **File Wizard** now appears. You should now select the “**Empty Java File**” option and click **Next**. Now you will be asked for the name of your program as well as its location. You should now use the browse button in the location field to browse to the location of the **Lab1** folder you just created on X: drive. Now, for the name of the file, you should type in **HelloWorld.java** and then click **Finish**.



The editor window should now be opened with the text – **HelloWorld.java** across the top as shown above. The cursor is at the top left of the code editor window where you will type in the “Hello World!” Java program code discussed earlier.

When you have typed the program in **exactly** as indicated earlier you should save it using the save icon on the IDE.

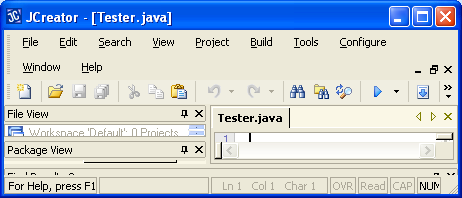
**Compiling and Running your First Java Program**

**What is Compiling?**

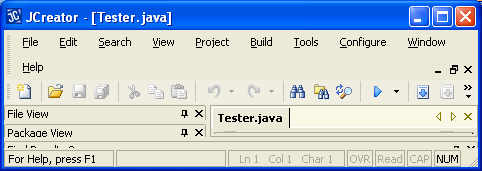
Compiling code basically involves **verifying that the program you typed in is free from syntax errors**. A syntax error is a violation of the grammatical rules of the programming language and we will look at some potential Java syntax errors shortly. If the program has any **syntax errors**, the compilation process will detect them and tell you about them in the **“Build Output”** window at the **bottom** of the IDE window.

If compilation was successful then the compiler **can translate your program into machine code** which can be executed and you should then see the output of your program displayed in a new window called the **output window** as indicated below.

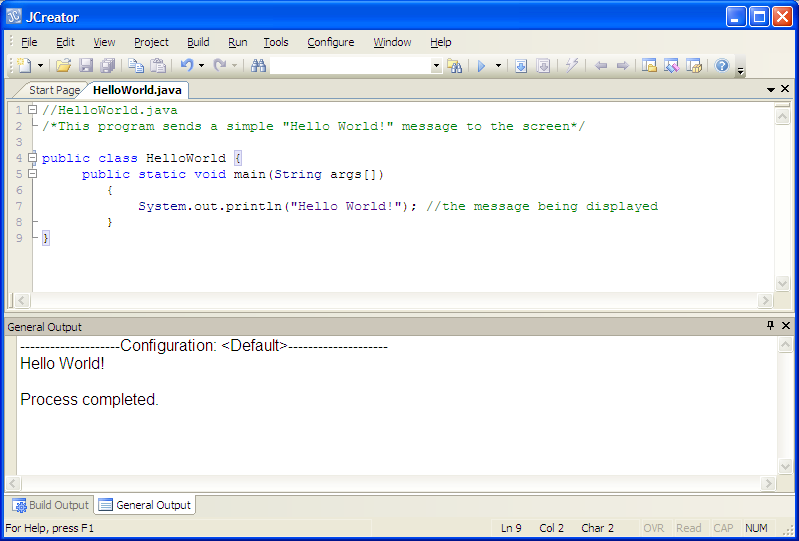
A program that contains even a single syntax error **cannot compile** fully and so **cannot be run**.

**Compile** the application by using the **build file** icon 

If your program compiles successfully, you should see the message “**Process Completed**” in the “Build Output” window at the bottom of the IDE window.

Now you can run the application by using the **execute file icon**  in the IDE .

If your program runs successfully, you should see the following displayed in the “**General Output**” window. Note that the text “Process Completed” appears automatically here to indicate that the program has terminated correctly.



If your program has any errors or warnings, have a look at the editor window again and check to ensure that the code is **exactly** as indicated earlier in this handout. If you spot any differences, correct them and **compile and run again** until the program is syntax error-free. **Always make sure you re-compile if you make any changes to your code**. **Typos** are going to be by far the most common source of syntax error at this stage.

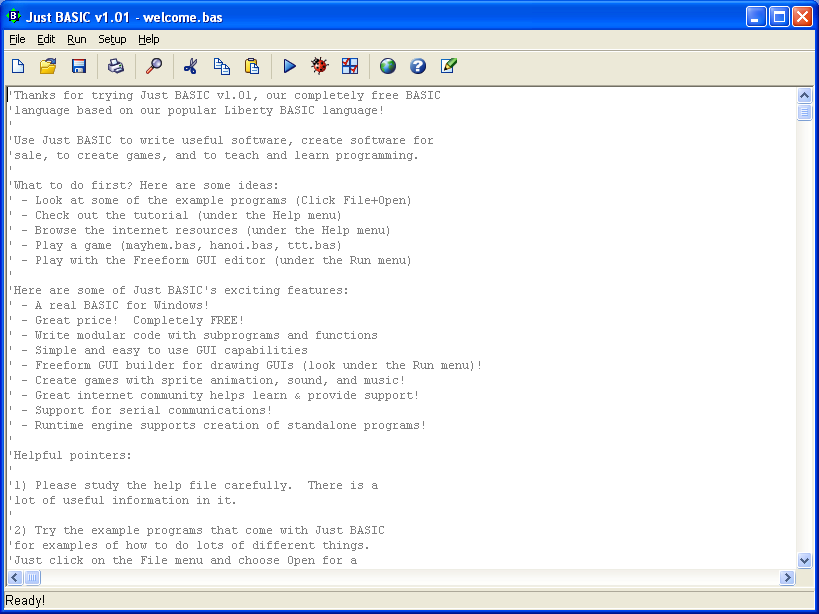
**If you see the output indicated above then congratulations - you are now a Java programmer!**

**Quitting JCreator**

Now that you have written a Java program, compiled it, executed it and saved it, you deserve a well-earned rest, for a few seconds anyway! To exit from JCreator just click the **X icon** in the top-right of the JCreator IDE window. This closes the JCreator application and any files which are open.

**Opening the Previously Saved “Hello World!” Program**

Start up JCreator again now. To open up a previously saved program, such as **HelloWorld.java**, do the following:

• click on the **Open File** icon 

• In the window that appears, navigate, if necessary, to the location of the **Lab1** folder on your **X: drive** where you should see listed the file called **HelloWorld.java**

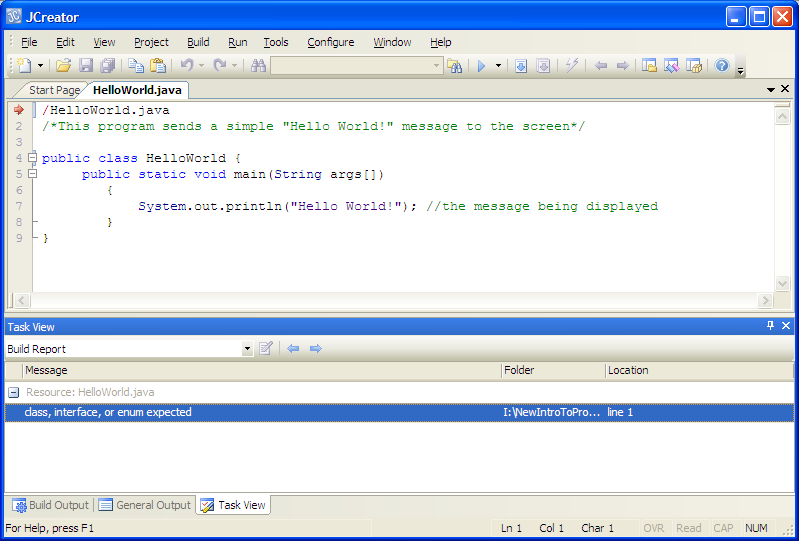
• To open this file in JCreator just **select the file** and click **Open**

• The file should now open up in the editor window

**Getting used to Syntax Errors**

As a programmer, it is **essential that you see errors as soon as possible**. Many of you may have typed in the first program perfectly and seen none. Therefore this section deals with putting some syntax errors into the program so that you can see how Java deals with them. This is something we will do throughout the course so that you become familiar with syntax errors of various types.

In your HelloWorld.java program, **take out one of the forward slashes** at the start of the single-line comment as indicated below and recompile.

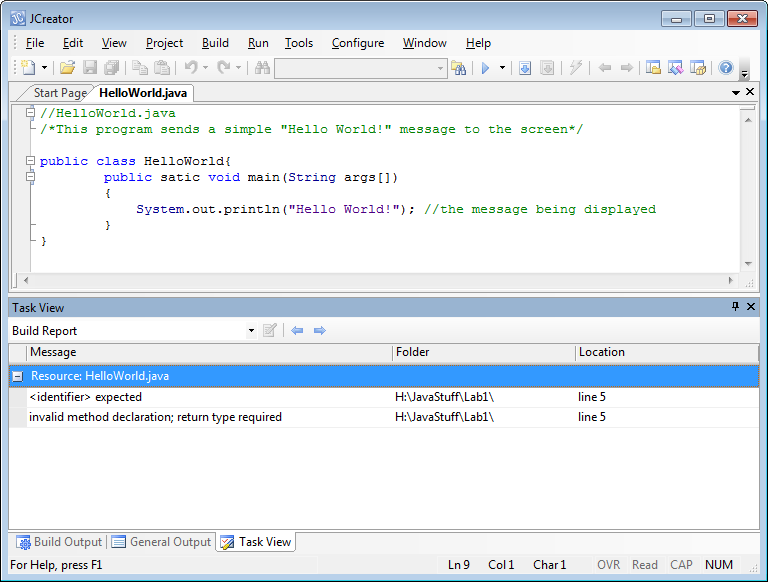


You will see that the error message “**class, interface or enum expected**” appears in the “Task View” window at the bottom of the IDE. Now you should **double-click** on this message and you will notice that a red arrow points to the line containing the syntax error as indicated above.

So the compiler is telling us it has encountered a problem with the first line of code and therefore cannot create an executable that we can run. Note that the **error message is quite cryptic** here and doesn’t actually tell us a whole lot about the real problem that exists i.e. that we have not written our single-line comment correctly.

We must now analyse the line of code where the syntax error occurred and see if we can solve the problem. Having analysed the program earlier, we already know that a **single-line** **comment must begin with //** so we can sort this error easily enough. Fix it up now and recompile.

As an example of another syntax error, **try changing the word “static” to “satic”**. Now, try to compile the program.



Again, the compiler tells you that a syntax error has occurred.

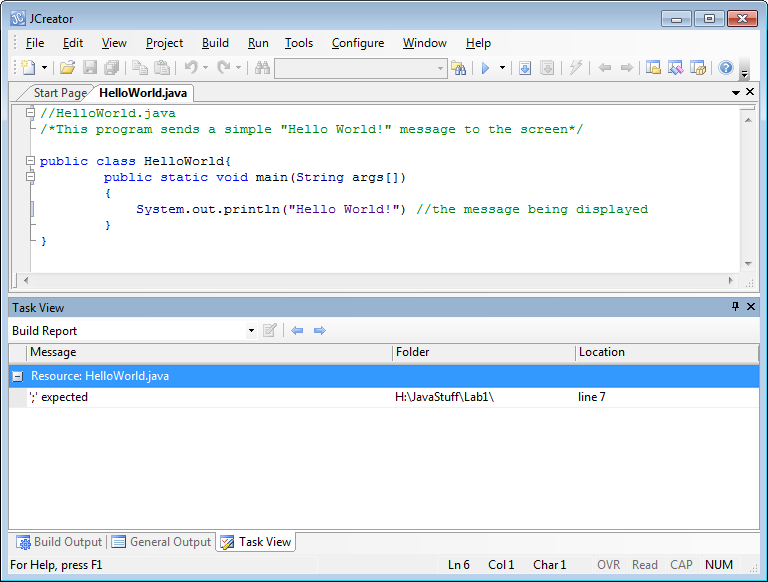
This time there appears to be “2 errors for the price of 1” – this occurs quite often when compiling in Java. One error can produce several error messages. The first error message indicates that an “**identifier is expected**” (line 5) and the second indicates that there is a “**invalid method declaration; return type required**” (line 5).

Now the real problem here is on line 5 where we have misspelt the keyword static. This does constitute an “**invalid method declaration**” so the second error is partly correct, although there is a return type in the line of code (**void** is the return type here). The first error message is not helpful to us here though.

This is always an issue with compilers – they are **by no means perfect** and they do their best to locate syntax errors and report them but their **associated error messages are very often cryptic** and, when an error is located, they **often signal extra error messages in locations where no error exists** – you have been warned! Therefore, it is **very important that you get to know the syntax rules of the language very well** so that you can decipher the error messages and correct them properly.

Fix up this error now and recompile.

Another opportunity to see a syntax error in this program would be to **leave out the semi-colon** at the end of the println() method statement and recompile again:



Notice this time that there is a **more useful compiler error message**. It says “**; expected**” i.e. a semi-colon is expected on line 7.

As you go on you will see more and more syntax errors – it is **vital that you see them so that you can learn how to fix them**. As you get more comfortable with the syntax of the language, you will get better and better at fixing these errors. I still very rarely write a program without making a few syntax errors – and that’s after 24 years of coding! All that matters is that you can fix them when they do occur.

**Syntax Colouring**

One of the most useful features available to a programmer is the colour-coding associated with the editor window. Different parts of the program have their own unique colours, which you may have noticed already. For example, when you add **comments** to the program, they all appear in **green.**

Likewise, you may have noticed that the **keywords** public, static, void and class all appeared in **blue**. So without even attempting to compile a program you can immediately tell if you have a valid keyword in your program based on the colour. For example, if you called the keyword “satic” by accident it would show up in the editor window in black rather than blue. Likewise, removing one of the forward slashes from the single-line comment causes it to appear in black rather than green.

So, without even attempting to compile your program you can often tell that there is something wrong with it through the **syntax colouring feature**, which is very useful indeed. This is where an IDE has a significant advantage over using regular text editors such as notepad which are literally “black and white”. The colouring feature also makes the program code more professional looking and nicer to look at.

You can see from the above example also that **text messages**, when written correctly in a program, appear in **purple**. Leave out the quotes and see how the colour changes, immediately indicating a problem.

My code screenshots throughout this module will indicate a slightly different syntax-colouring scheme from the one that you will see by default in the labs. This is nothing to worry about at all, indeed you are welcome to alter the syntax colouring scheme to suit your own needs in the labs or at home. I will send you all on an email that shows how you can change the syntax colouring scheme to reflect the one I have used in my lab sheets, if you wish to do so.

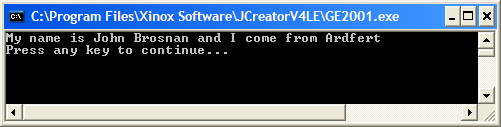
**Exercise 1**

At this stage you have everything you need to write your own Java programs which output simple plain text messages to the screen.

Write a program which outputs the following message to the screen:

**My name is <*your name*> and I come from <*your location>***

So if I was writing this program it would run as follows:

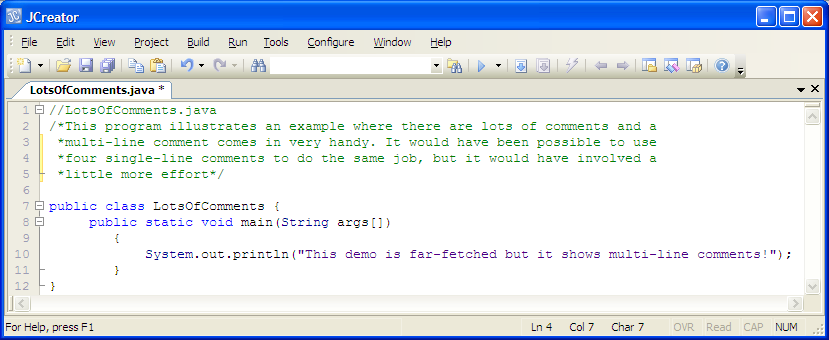


Don’t forget to put a **single-line comment** for the file name and a meaningful **multi-line** **comment** at the top of your program to indicate what the purpose of your program is. Save it as **Exercise1.java** in the folder **Lab1** on your X: drive and then compile and run it.

Note here that if you set up your IDE to display the **console output window** separately, as I have done, the “**Press any key to continue …**” message appears automatically once your program has completed execution. If you use the default “General Output” window, as we have been doing all along, then your output will appear in this window, just below your code. All my screen-shots of output in these lab sheets will refer to the console output window indicated above but the results should generally be the same. You don’t have to use the console window for your own program runs, I just personally prefer it.

**Long Comments in a Program**

If you have long comments in a program, **just use a multi-line comment**.



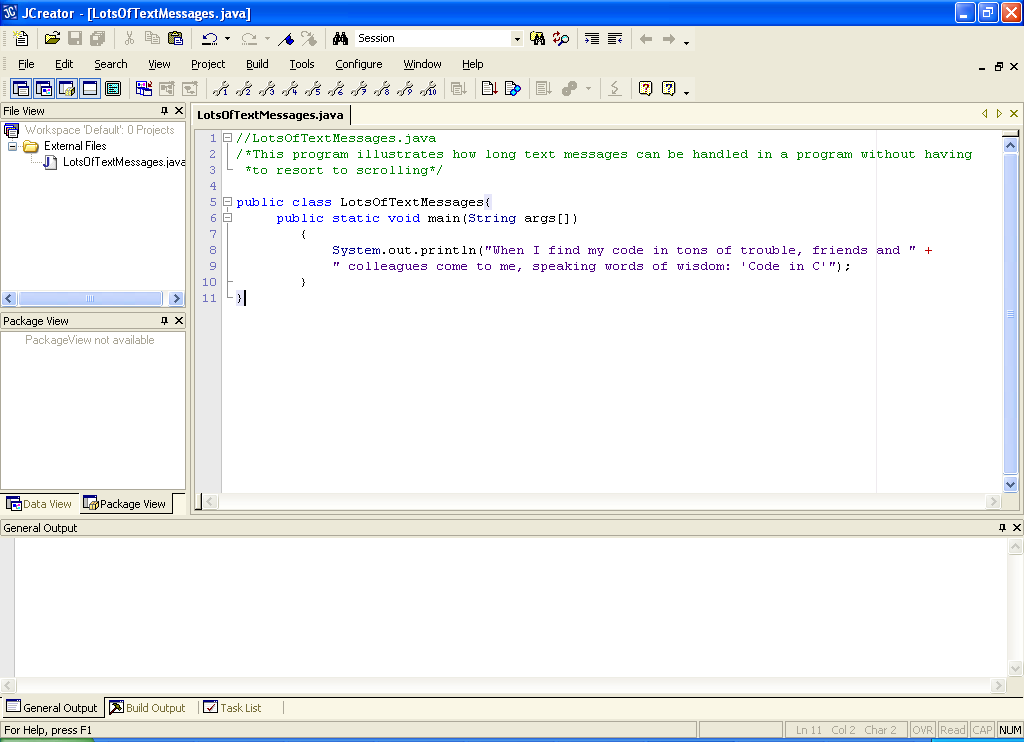
Of course, it would have been possible to **put the entire text for the comment** in the program above on the one line but then in order to read the contents of the comment, you would need to use the scrollbar to **scroll** over, which is possible but **not very efficient**.

When reading code you should rarely, if ever, have to use the scrollbar. It is **good programming practice** to keep the contents of a comment (and all lines of code in general) within the width of the code editor window.

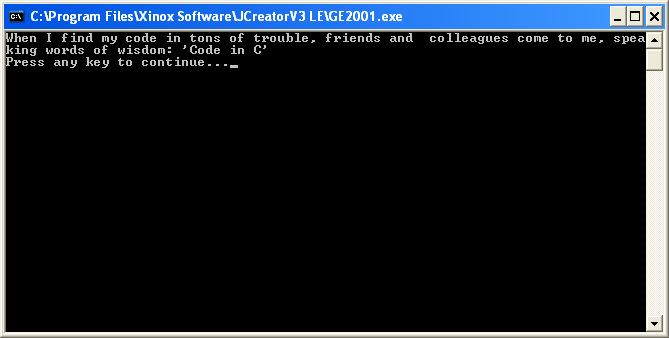
You don’t need to type in the program above unless you want to for practice.

**Long Text Messages in a Program**

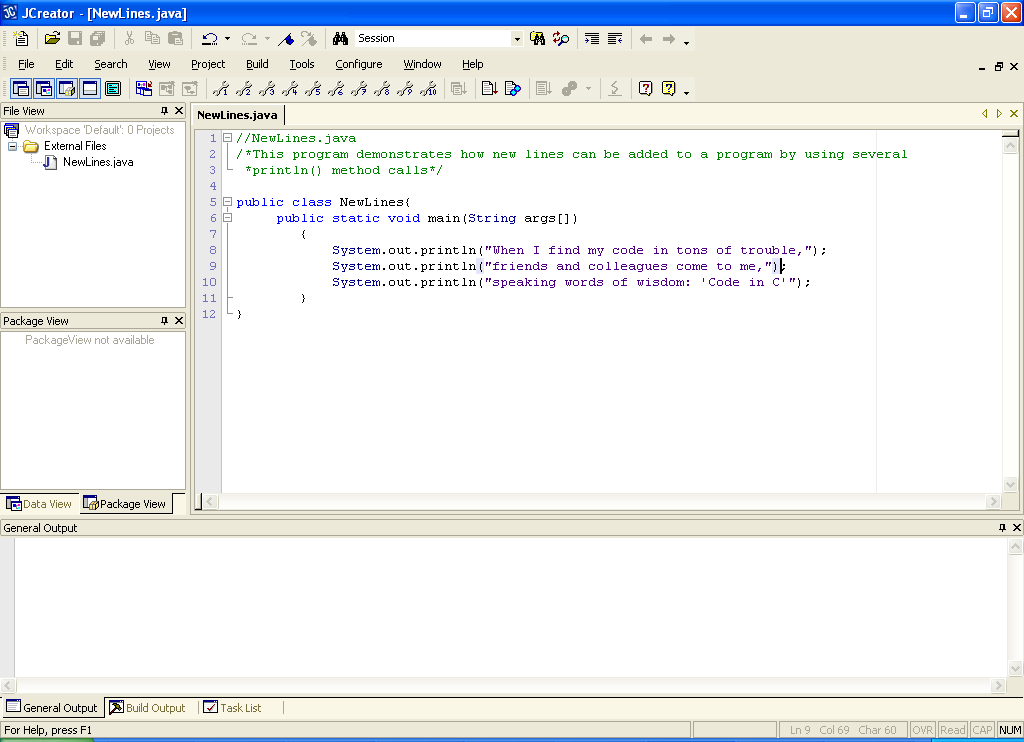
In Java, long text messages can be handled using a **special operator**. The program **LotsOfTextMessages.java** below demonstrates this operator. At the end of the first section of text within the println() you can use the operator **+** to allow you to **add the text on the following line to the current line.** This is called the **string concatenation** operator as it concatenates (joins) two strings together.



Type in the program above, save it as **LotsOfTextMessages.java**, compile and run it. In the console window the program runs as follows (if you are viewing the output in the “General Output” window you will see a slightly different result to this but don’t worry about that).

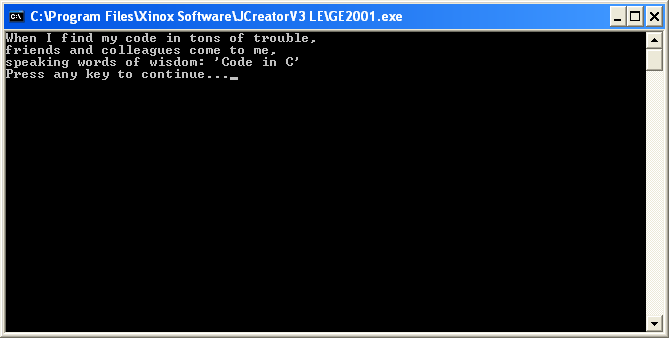
****

Notice that the output here isn’t really ideal. For one thing we would prefer the word “speaking” not to be split up in the way it is (this may not be an issue in your case). However, more importantly, as this is a poem of sorts, we would like to add **new lines** to our program so that each line of the poem automatically wraps when it is displayed on the output window. This can be easily **achieved by using several println() method calls** as follows – this is because each println() call places the cursor at the beginning of the following line when it has displayed its text.



Type in the code for this program, save it as **NewLines.java**, compile and run it.

This program runs as follows – no splitting of words anymore and it reads like a poem!



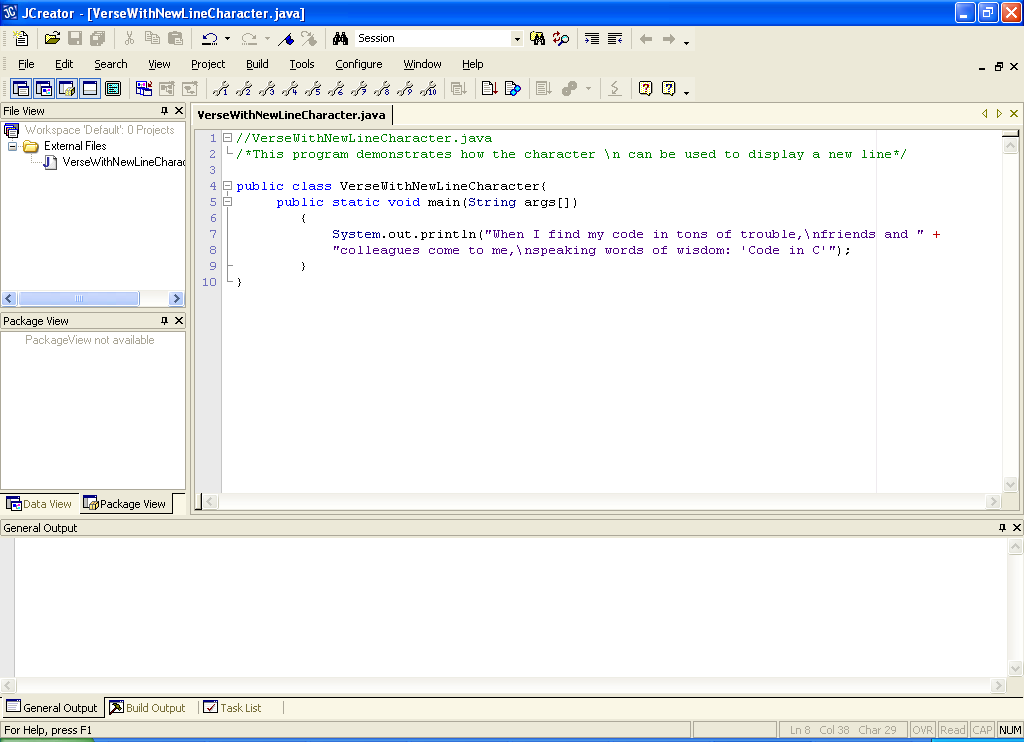
**Using \n for New Lines**

Notice in the last example that we used multiple **println()** method calls to achieve our goal. However, there is an alternative, more efficient, way of doing this. One of the key things you will notice as we continue through the course is the fact that there is **almost always an alternative way of tackling a particular problem**. There is always an emphasis on **efficiency** – trying to write programs in the most efficient way possible. You will notice that in programming, there is **often more than one correct solution** to a given problem.

There is a general need for **efficiency** in programs so that they may execute as quickly as possible. In general, the more code that a program contains the longer it takes to execute. You should always aim to code your solutions so that they contain the minimum amount of code required.

The alternative way of displaying the poem is to take advantage of a **special Java character** called the newline character **\n**. When a newline character is displayed it won’t literally display the characters \ and n, instead it has the effect of bringing the cursor immediately onto the next line.

The beauty of the \n solution is that **we do not now need to make multiple calls to the println()** method to achieve our goal. Now the solution becomes:



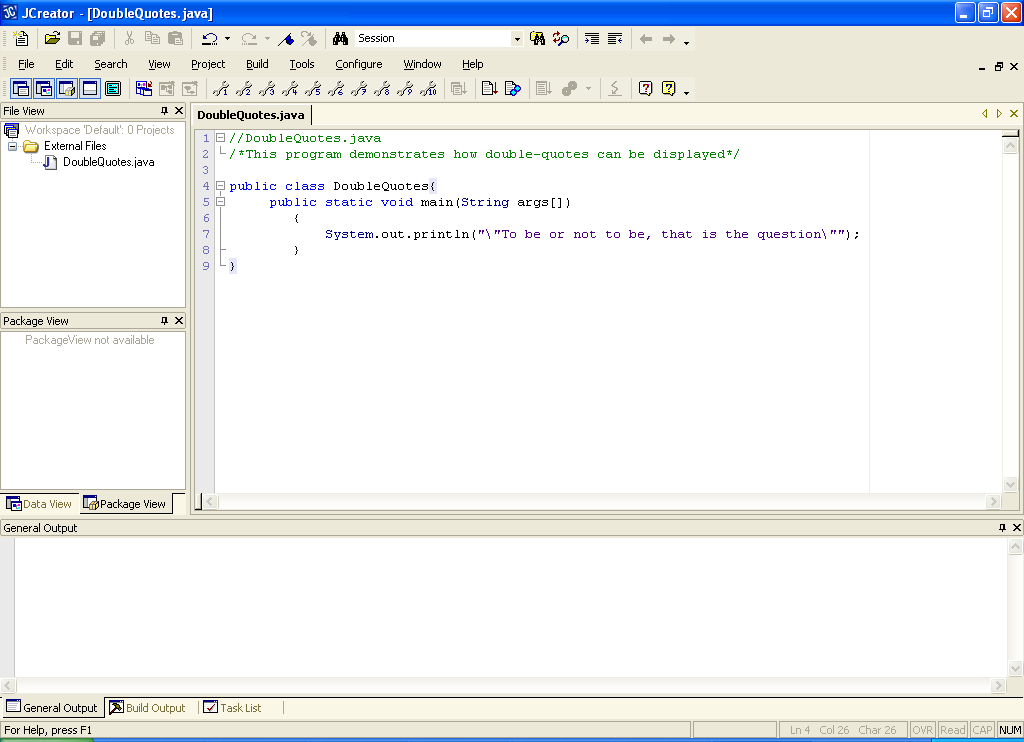
Notice that the newline character **must be placed within the double-quotes of the println**() just like all the other characters. Note also that at the end of the first line of text is the **+**  which is used to join the text on the next line to the current line.

Now take the existing file NewLines.java and save it as **VerseWithNewLineCharacter.java**. Now make the necessary changes to the existing code, compile and run as before. It should produce exactly the same output as NewLines.java

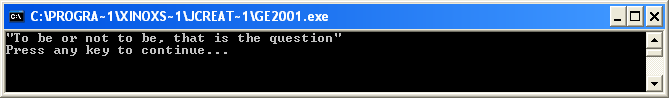
**Displaying Double-Quotes**

Placing text including double-quotes directly within a println() method is a **syntax error** as a **conflict** occurs. You should prove this to yourself now by **modifying the file VerseWithNewLineCharacter.java** above to put double-quotes at the beginning and end of the text message.

To get around this conflict, double-quotes are represented by the special character **\”** This is illustrated in the following example:



The program then runs as follows:



For practice, create the file **DoubleQuotes.java**, type in the code above, compile and run the program.

**Other Special Characters in Java**

Apart from \n and \”, there are some other special characters in Java. These include

\\ – to display a **backslash**

\t – to display a **horizontal tab**

\r– to display a **carriage return** (in the console window this puts the screen cursor at beginning of current line without advancing to the next line – characters displayed after the return overwrite characters previously output on that line – this character behaves like a \n in the “General Output” window however)

**Exercise 2**

Using what you have learned so far, write a Java program that produces the following output **exactly** **as indicated** in the screenshot below. You **should only use a single println() method** here along with the various special characters. Don’t forget the **comments**! Save your program as **Exercise2.java** in the Lab1 folder.

