**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

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

**Practical 1 – Quick Review of Java Basics**

Welcome back! Hopefully you enjoyed the break and are ready for some more programming. This module continues on from the “Introduction to Programming” module before Christmas and aims to first of all **review the basics of structured programming** covered in that module. It then moves on to **cover the remaining structured programming principles**. The module also aims to expand your knowledge **of object-based programming** as well as introduce some very basic software documentation through the **Unified Modeling Language** (UML).

As always with programming, the main thrust of the **course 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.

**Course Outline**

•Review of structured programming basics

•GUI input and output

•Methods and Arrays

•Input Validation Algorithms

* Simple GUI components
* Using the API documentation
* Introduction to user-defined Classes
* Documentation through UML

**Exams**

It is expected that you will be given **3 continuous assessments** during this module, **accounting for 50% of the total marks** for the module. These assessments will be practical in nature where you will have the computer and a Java IDE/compiler available to you and **all of the marks will go for your coding ability**.

The **final exam** in May will be **worth 50%** 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**. You need to get at least 40% overall to pass the module and at least 30% in the final exam itself. You also need to attend at least 75% of the labs.

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

•The ability to code, debug and test relatively simple Java programs involving GUI components.

• The ability to translate thoughts on paper into actual "real-life" Java programs.

• Familiarisation with the entire principles of structured programming which can be used to learn other programming languages much more easily.

• Recognition of the importance of, and the ability to code input validation routines to ensure more robust systems.

• A knowledge of the fundamentals of object-based programming.

• The ability to examine the Java documentation for predefined classes and use methods within those classes.

• A very basic understanding of UML.

**Getting Started**

Some of you may have transferred from other courses and some may be Erasmus students, who have not taken the 1st semester “Introduction to Programming” module. In any case, many people will be rusty on their Java after the long Christmas break and there are also those who struggled with Java in semester 1. Therefore the first 2 lab sheets are designed to very quickly review the material covered in semester 1. Especially if you are a newcomer, consult these lab sheets for more detail. I will give all newcomers access to these semester 1 lab sheets on X: drive, let me know if you don’t have access to them.

**(Re)introducing Java**

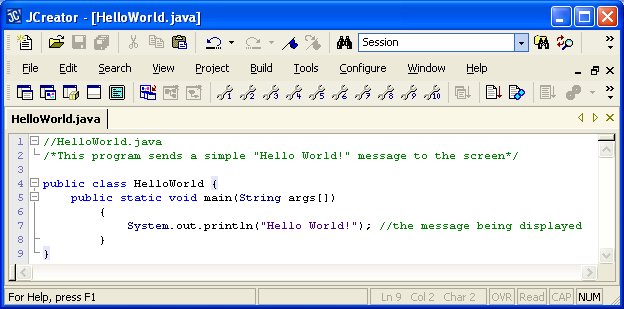
Java is a **structured**, **high-level**, **object-oriented** programming language. It was first released in May 1995 by **Sun Microsystems** in the USA. It is now owned by **Oracle**.

**What is an Object-Oriented Language and why use one?**

There are two main types of programming language – those that are **procedural languages**, such as Fortran, Pascal and C, and those that are **object-oriented**, such as Visual BASIC, C++ and Java. An object-oriented (OO) language is one that is **oriented towards the creation of** **objects**. There are **several advantages associated with OO languages over procedural ones**, especially when it comes to the design of complex software systems, which we will not go into here - except to say that **they promote software reuse** (more on this next year in OOP2 and OOP3).

**The Basic Java Program**

**Java Code**:



**Quick Analysis**

// signifies a single-line comment

/\* \*/ signifies a multi-line comment

Comments can appear almost anywhere within a Java program

**public**, **class**, **static** and **void** are Java **keywords** (which were pretty much glossed over in semester 1)

A Java **class** is like a **container** for your code. Here the class is called **HelloWorld**.

Classes are delimited by **curly braces**. Every Java program needs a class in order to be useful. By convention, class names begin with an uppercase letter and are **case-sensitive**.

Within our Java classes there will always be (for now at least) a **method** called **main**(). You can think about the main() as being like a container for your code also as this is where most of the instructions for your program will go at the moment. All desktop Java programs need a main() in order to run. Methods are also delimited by **curly braces**. By convention method names begin with a lowercase letter and are **case-sensitive**.

In Java the main() method has to have the signature

**public static void main(String args[])**

the only thing that can be changed here is the word **args**, but most programmers leave it as is.

This program has one instruction. This is the call to System.out.**println**() to display the message to the output window. That line of code is a **statement**, which is terminated with a **semi-colon.** This was one of the most commonly used methods in semester 1.

**Getting Started on the PC**

Switch on your machine if it is turned off and log in to the network.

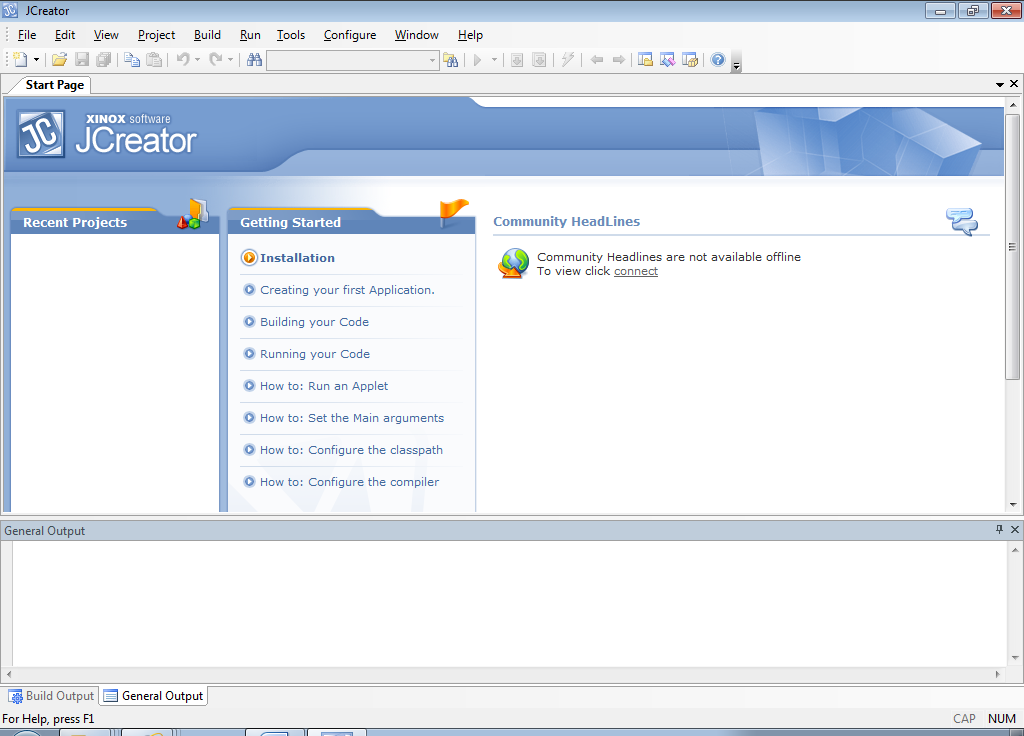
The following sections are for any newcomers who have not used JCreator before. Most of you can just go to the “**Setting up your Folder Structure**” section now and then **skip to Exercise 1** at the end of page 8**.**

**Getting into JCreator LE**

Go to the following location on the system:

**All Programs**->**Java**->**JCreator Pro**->**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.

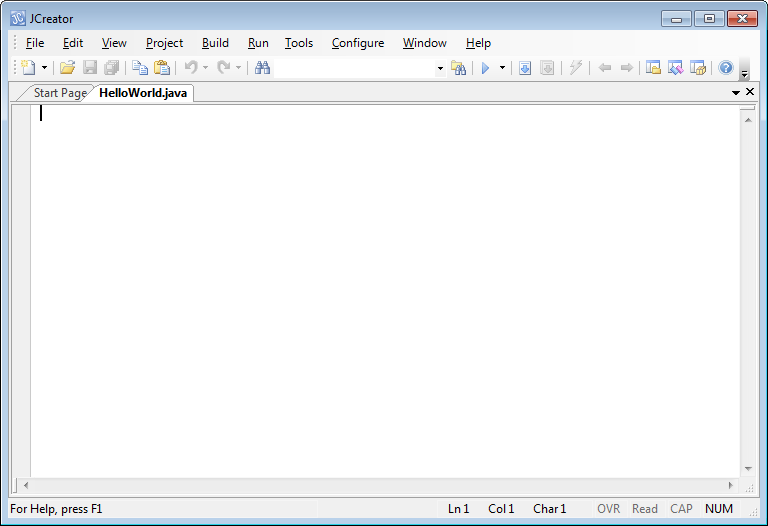
**Setting up your Folder Structure**

Create a folder called **OOP1Stuff** on your X: drive and in 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, choose **File** from the main menu and then **New** and then **File**. Better again, you can just click the **New** **file** **icon** immediately above the File menu.

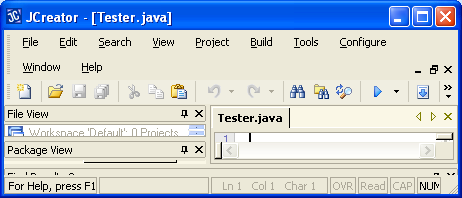
A **File Wizard** now appears. You should now select the “Empty Java File” option and click **Next**. Now you will be asked for the name of your program and as well as its location. You should now use the browse button in the location field to browse to the location of the **Lab1** folder you just created.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



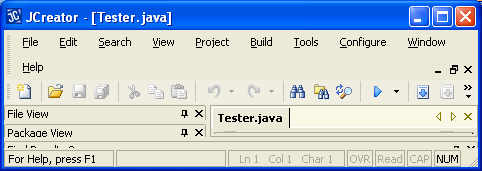
The edit window should now be opened with the text – **HelloWorld.java** across the top as shown above. The cursor is at the top left of the code editor window where you will type in the Java program code discussed earlier.

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

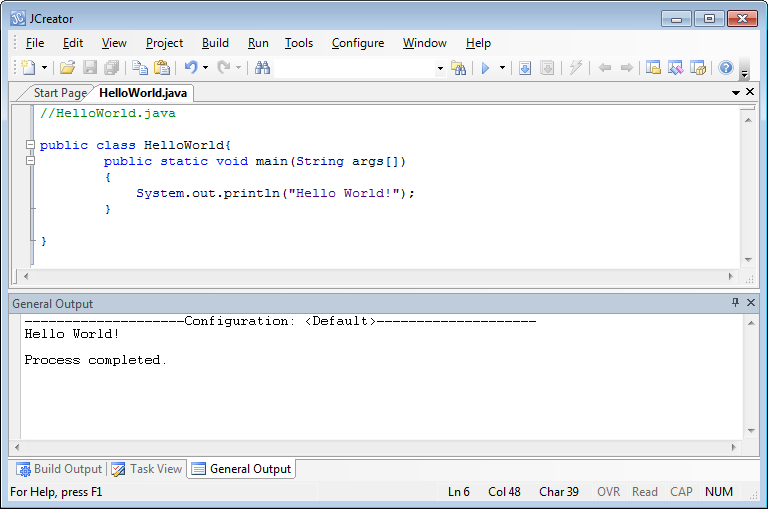
**Compiling and Running your First Java Program**

**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 this handout. If you spot any differences, correct them and **compile and run again** until the program is syntax error-free.

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

**Getting used to Syntax Errors**

Any newcomers to Java should now mess around a little with the HelloWorld.java program by modifying the code bit by bit and recompiling with each modification. This is just to give you an idea of the kinds of error the Java compiler can throw up. Some of the things you can do are

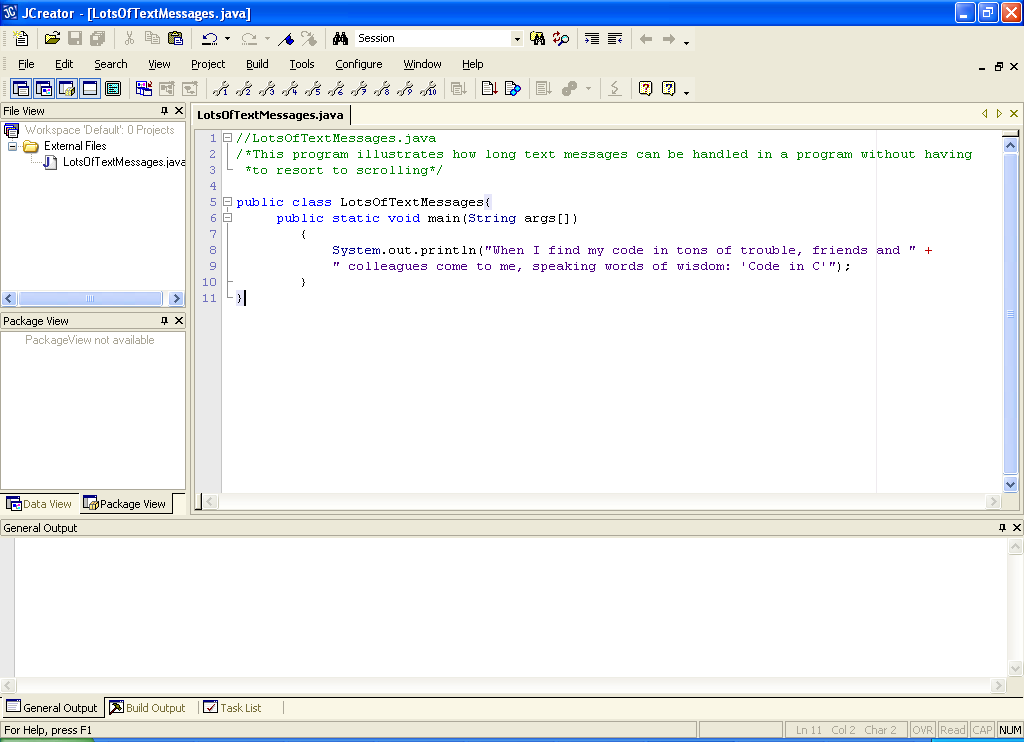
* change the name of the class to helloWorld with a lowercase h
* leave out the opening curly brace on the class
* change the keyword void to voit
* change the multi-line comment to end with /\*
* leave out the semi-colon on the println() statement
* leave out the opening double-quotes when displaying the message

**Syntax Colouring**

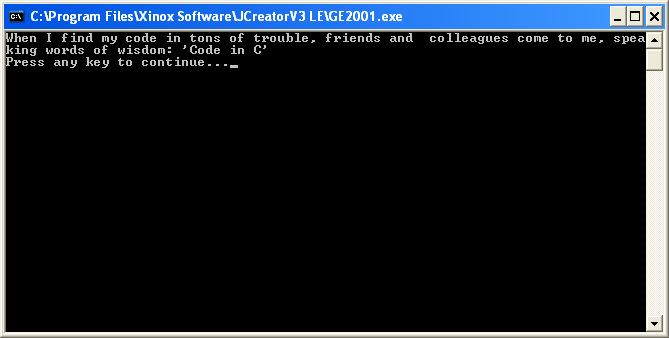
You can see from the screenshot above that certain entities appear in certain colours within the editor window. Comments are **green**, keywords are **blue**, classnames and method names are **black** etc. This is called **syntax colouring** and makes programs easier to read and debug.

**Long Text Messages in a Program**

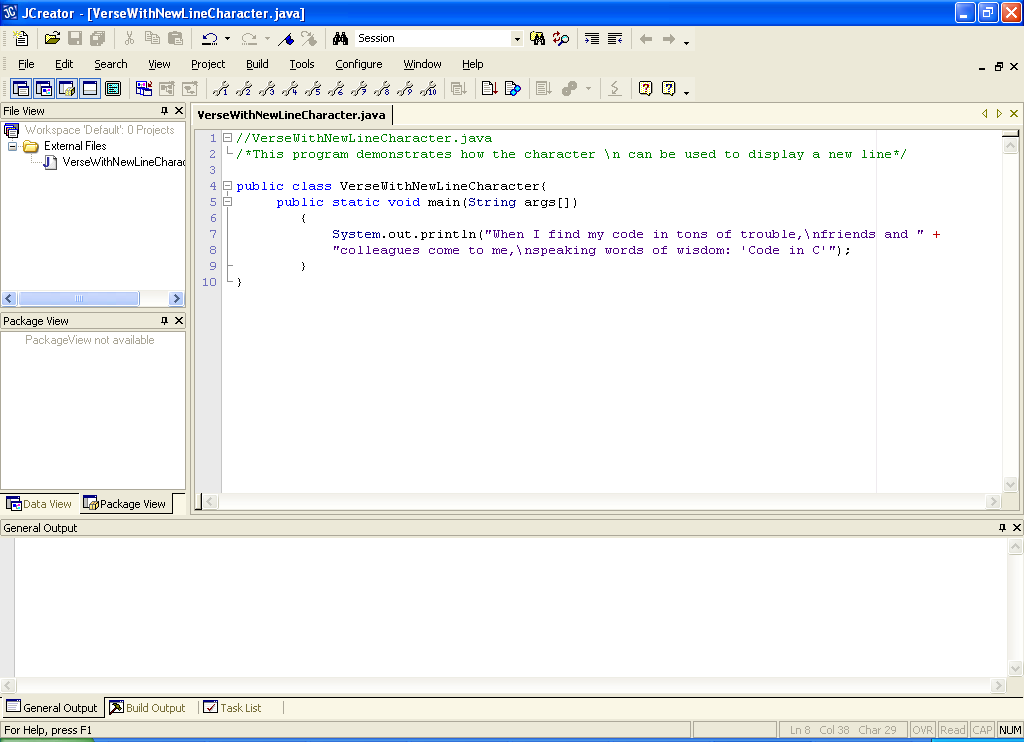
In Java long text messages can be handled using a **special operator**. The program **LotsOfTextMessages.java** below demonstrates this operator. At the end of the first section of text within the println() you can use the operator **+** to allow you to **add the text on the following line to the current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



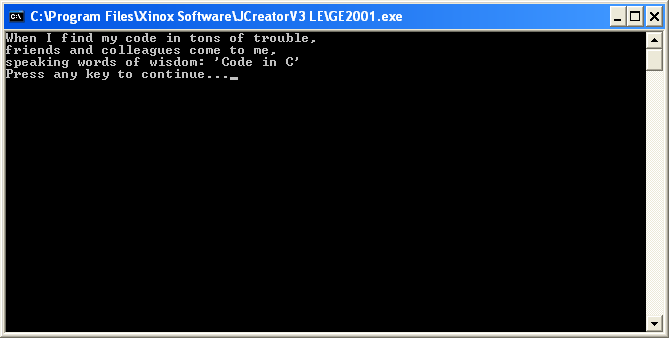
Although this compiles fine, it runs as follows:

****

The program works alright but the output could be displayed better, especially as this is a sort of poem. We want **new lines** to help us here to split up the output over a few lines. Java has a special character for this – the **\n** character - as used below:



Now the program runs as follows:

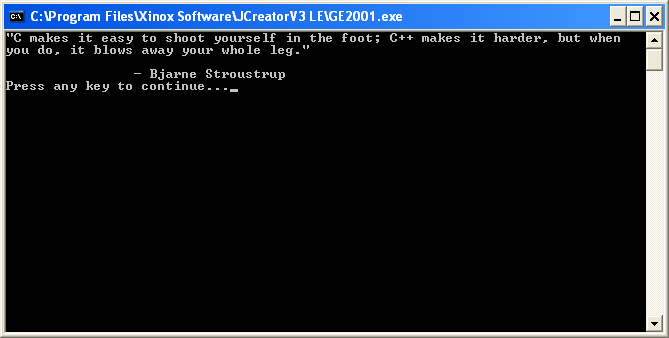


**Other Special Characters in Java**

Java also has special characters for displaying **double-quotes** - **\”**, **tabs** - **\t** and **backslash** itself - **\\**

**Exercise 1**

Write a Java program called **Exercise1.java** that uses special characters and a **single** **println**() to display the following message to the screen:

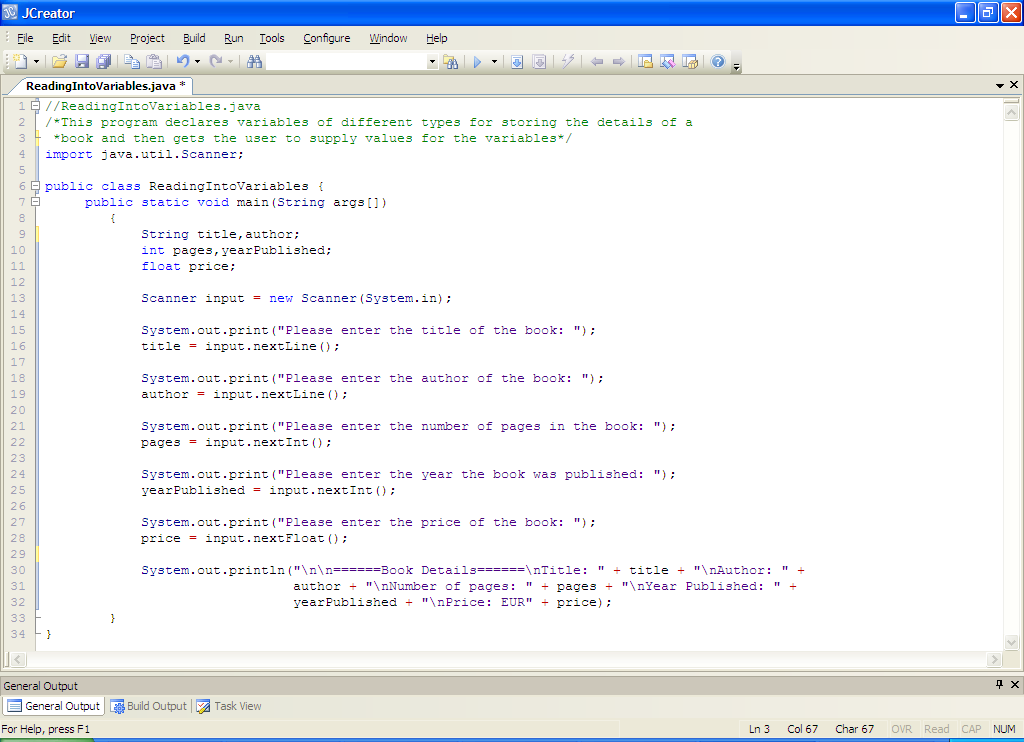


Don’t forget to put **comments** at the top of your program to indicate the name of the program and what the purpose of your program is.

Note here that if you set up your IDE to display the **console output window** separately, the “**Press any key to continue …**” message appears automatically once your program has completed execution. If you use the default “General Output” window, then your output will appear in this window, just below your code. All my screen-shots of output in the first 2 lab sheets will always refer to the black console output window indicated above but the results should be the same.

**Variables and Keyboard Input in Java**

Java supports many types of variable for data storage but the 3 we focused on mostly in semester 1 were **integer** variables (**int**), **floating-point** variables (**float**) and **String** variables (**String**). The following program illustrates their use:



**Quick Analysis**

• The program requires reading in information from the user. We use a class called **Scanner** to facilitate this. In order to use this class we must **import** it. Generally, classes that a Java program requires have to be imported at the top of the program, before the class definition. Classes belong to **packages** in Java. Scanner belongs to the **java.util** package. As it happens, this is the only class we ever needed to import in semester 1, as all other classes we required (such as **System**) were automatically imported as they belonged to a special package called **java.lang**.

• The 5 variables are declared. Variables must be declared before they are used in a program. By convention they begin with a lowercase letter. Variables can be given values when they are declared also e.g.

**int pages=0,yearPublished=0;**

would **initialize** the variables here with the value 0.

• The code

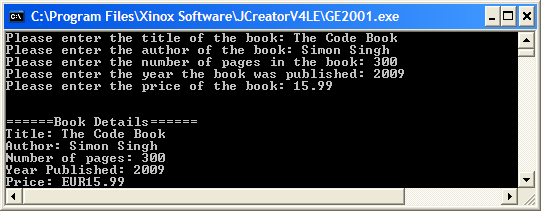
**Scanner input = new Scanner(System.in);**

Appears in almost every program in semester 1, since almost all programs required user input. The code makes a connection to the keyboard (**System.in**) using the Scanner class we previously imported and will use a reference (input) to the Scanner object created (through the **new** keyword) to allow us read data from the keyboard.

• Next we have the prompts to the user for input – **print**() is used to keep the cursor on the same line as the prompt. Following each prompt is a call to a method to read in the type of data expected. So the method **nextLine**() is used to read in the string (text) information, **nextInt**() is used to read in whole number information while **nextFloat**() is used to read in numbers that might contain a decimal point.

• The program ends by just displaying back the values that the variables now hold.

The program would run as indicated in the following sample screenshot:



**Typing in the Code For the Program**

It would be especially beneficial for any newcomers to Java to type in the program above just to get used to the syntax of the Java language. Save the program as **ReadingIntoVariables.java** in your Lab1 folder, compile, debug and run it.

**Basic Output Formatting**

We used a method called **String.format()** to format the output display in semester 1. This method uses what is called **format specifiers** to decide how the output will get displayed. The format specifiers are:

**%s** for formatting strings

**%d** for formatting integers

**%f** for formatting floating-point numbers

The basic format specifiers above are then **modified** through the use of flags and numbers in order to decide exactly how information will display e.g.

**System.out.println(String.format(“%-25s%-30s”,name,address));**

Would cause the name and address to be displayed as follows:

**Joe Bloggs Anytown, Anywhere**

**|-------25 wide-------|-------------30 wide-----------|**

The – before the 25 and the 30 mean **left-justify** the information. If this were absent, the result would be right-justified instead

**Joe Bloggs Anytown, Anywhere**

**|-------25 wide-------|-------------30 wide-----------|**

Another example would be:

**System.out.println(String.format(“%-10d%-5.2f”,stockID,price));**

Would cause the stock ID and price to be displayed as follows:

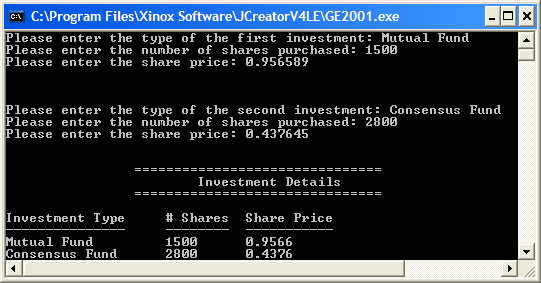
**243578 23.87**

**|--10 wide--|---5 --|**

The price is displayed to 2 decimal places here through the**.2** after the number 5

**Exercise 2**

Write a Java program that prompts for and reads in the details of 2 types of investment for the share portfolio of an investor. The details to be entered are the investment type, number of shares purchased (a whole number) and the current share price. Once the details are entered, they should be displayed nicely as indicated in the sample screenshot below, using the **format**() method. Try to use a **single println**() for display here. Note that the share price is to be displayed to **4 decimal places**. Also note that for the “Investment Details” banner I have used **tabs** rather than format(). Try to do this also for practice.



One thing to watch out for in your coding here is to make sure that you make an extra call to **input.nextLine()**, just **before** you read in the second investment type. Otherwise you will have a **logical error** in your program. Anytime you have string input following numeric input using a Scanner, you must take this precaution.

**Arithmetic in Java**

Java has 5 arithmetic operators to allow basic calculations. These are +, -, \*, / and %. The last one is called the **remainder** **operator**. It gives the remainder upon division of one number by another e.g. 17 % 3 is 2. One other thing to watch out for is that the division of one integer by another gives an integer, chopping off anything after the decimal point e.g. 15/4 would give 3 and not 3.75. This can be used to your advantage in various situations. If you wanted the accurate answer, you would have to say 15f/4 or 15/4f, effectively forcing one of the values to become a floating-point number through the use of the ‘f’.

**Some examples of coding some formulas:**

**E = m\*c\*c;**  (E = mc2)

**circleArea = PI\*r\*r;** (area of a circle = πr2).

Here PI would have been declared as a **constant** in the program as follows:

**final float PI = 3.142f;**

the use of the final keyword makes a constant out of a variable. Now its **value cannot change**.

**F = G\*m1\*m2/(d\*d);** ()

Notice that the **parentheses** are needed in the last example. Otherwise we will have a **logical error**. Likewise for the case below:

**averageOf3Numbers = (a+b+c)/3;** (average =

**Correcting to the nearest whole number**

In order to correct a value to the nearest whole number just use **String.format(“%.0f”,value)**

**Type-casting**

Type-casting can be used to convert a value to a different type momentarily. It has several uses e.g. if we wanted to find the whole part of a fractional variable x which holds the value 23.678. Here we could use type-casting as follows;

**int wholePartOfX = (int)x;**

now the variable wholePartOfX would hold the value 23.

It is often used in a situation where you have two integer variables where one is divided by the other, and you need to get an accurate answer e.g.

**average = sumOfValues/numberOfValues;**

assuming that both variables on the right are integers, the result of the division will be an integer. Therefore the average may well be inaccurate. To get the accurate result, we could type-cast either variable as a float as follows:

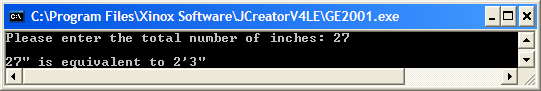
**average = (float) sumOfValues/numberOfValues;**

or

**average = sumOfValues/(float) numberOfValues;**

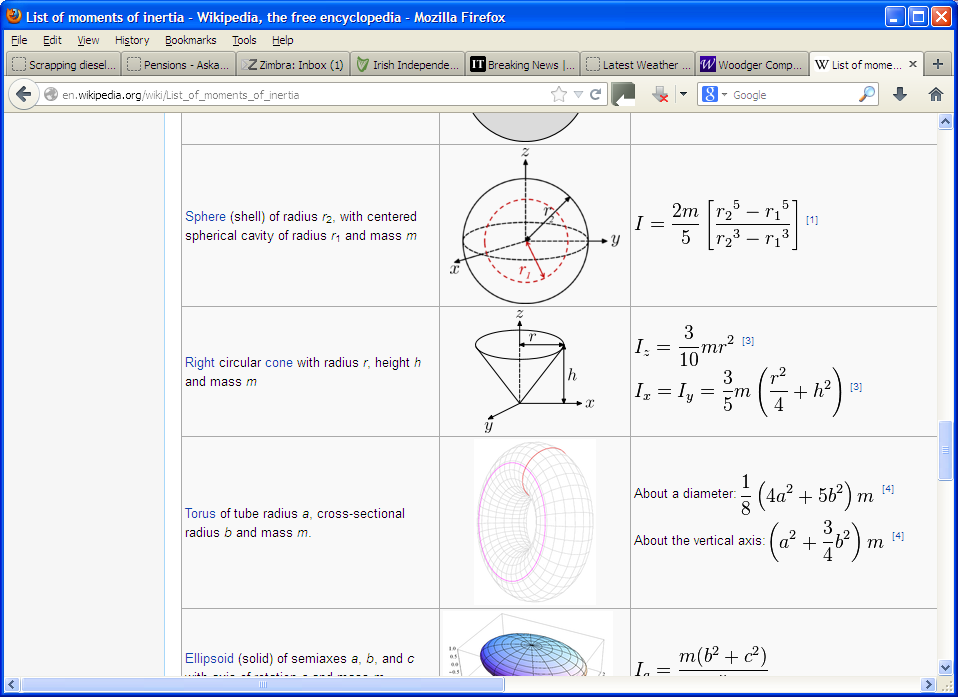
**Exercise 3**

Write a Java program that reads in a whole number of inches and converts this into feet and inches. Note that there are 12 inches in 1 foot. The program would run as follows:

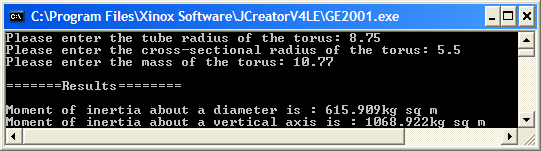


**Exercise 4**

The moments of inertia of a torus of tube radius a, cross-sectional radius b and mass m about a diameter and about a vertical axis are given by the formulae on the right below



Write a Java program that gets the user to supply the values for the tube radius, the cross-sectional radius and the mass of a torus and then uses the formulae above to determine the moments of inertia. Your program should display all results to **3 decimal places**.



**Decision-Making in Java**

The main decision-making structure in Java is the **if statement** and its various derivatives.

For example, if we want to test whether a number exceeds 100, the code might be:

**if(number > 100)**

**System.out.println(“\nThis number exceeds 100”);**

In this case, if the number turns out to be less than or equal to 100, then nothing happens as such. There is only output here is the number actually exceeds 100. This is called a **simple if statement**.

If we want to go a little further and check to see whether the number exceeds 100 and put out an appropriate message either way, then we can use an **if-else statement** as follows:

**if(number > 100)**

**System.out.println(“\nThis number exceeds 100”);**

**else**

**System.out.println(“\nThis number does not exceed 100”);**

Then there are occasions where there are many possible outcomes for a test and so a number of different sets of action might be needed. Here a **nested else-if structure** can be used, for example if we ask the user to enter their mark in an exam and then we wish to assign the appropriate grade based on this mark we could have the following code:

**if(mark >= 85)**

**grade = “A”;**

**else if(mark >= 70)**

**grade = “B”;**

**else if(mark >= 55)**

**grade = “C”;**

**else if(mark >= 40)**

**grade = “D”;**

**else**

**grade = “E”;**

All the structures above typically depend on the 6 **relational operators** in order to compare values. These are **==**, **!=**, **>=**, **<=**, **<** and **>** and can be used for int, char and float.

However, if we are comparing strings for equality, we must always use the **equals() method**, for example, if we want to check to see if the name of a person is “Tom Thumb” then we would have:

**if(name.equals(“Tom Thumb”))**

**System.out.println(“The name you entered was Tom Thumb”);**

**else**

**System.out.println(“The name you entered was not Tom Thumb”);**

So **name** above would have to be a **String variable**

Individual characters can be compared also in Java. Their ASCII codes are used behind the scenes for such comparisons. For example, if I wanted to check to see whether a certain character was a lowercase letter, I would use the code:

**if(character >= ‘a’ && character <= ‘z’)**

**System.out.println(“You entered a lowercase letter”);**

Note that the **logical AND operator** **&&** is used in the test above to combine 2 subexpressions. There are 2 other logical operators available in Java. These are the OR operator || and the not operator ! An example of using the OR operator is as follows:

**if(character < ‘a’ || character > ‘z’)**

**System.out.println(“You did not enter a lowercase letter”);**

As you can see above, individual characters can be compared in Java, but it is not actually possible to read in an individual character in Java. All we can do is to read in a string and then **extract** **the characters** from the string. We do this using the **charAt**() method. For example, if I have a String variable called address that holds the word “Tralee”, then I can access the first letter in the word as follows:

**System.out.println(“The first letter of the word is” + address.charAt(0));**

Similarly, to access the second letter, I pass the number 1 to the charAt() method etc.

This was one of the most commonly used methods in the module as there is very often a need to extract characters from a string.

Along with the charAt() method, another very useful method for dealing with strings is the **length()** method. This tells us how many characters a string contains e.g.

**String name = “Tom Thumb”;**

**System.out.println(“The number of characters in the name ” + name + “ is ” + name.length());**

**Exercise 5**

Health experts make use of a body mass index (BMI) value to determine whether a person is at the correct weight for their height. The formula for determining the BMI of an individual is as follows:

|  |  |  |
| --- | --- | --- |
| BMI = | Weight in Kilograms   (Height in Meters)² |  |

Once the BMI is calculated the following table can then be referenced to determine the status of the individual.

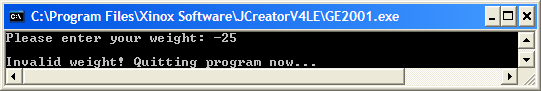
|  |  |
| --- | --- |
| **BMI** | **Weight Status** |
| Below 18.5 | Underweight |
| 18.5 – 24.99 | Normal |
| 25.0 – 29.99 | Overweight |
| 30.0 and Above | Obese |

Your task here is to write a Java program that firstly asks the user to enter their height in meters and their weight in kilos. Note that your program **should not accept invalid values** (i.e. **negative** ones) for either of these two quantities. In both these cases, the program should give an appropriate warning message and carrying out no extra processing whatsoever, simply **quitting the program at this stage**.

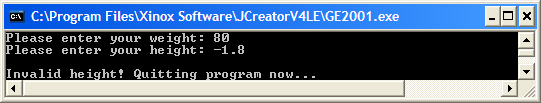
However, if both the weight and height data supplied are valid then the program should calculate the BMI value for the individual and use the table given above to generate an appropriate message to the user.

You should write your code so that it would run according to the following sample screenshots.

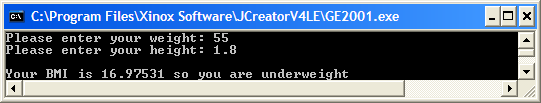
Run 1:



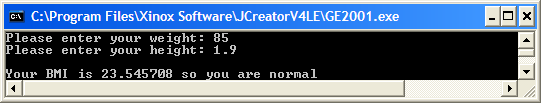
Run 2:

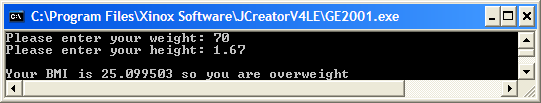


Run 3:

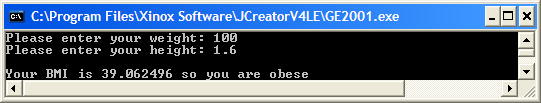


Run 4:



Run 5: 

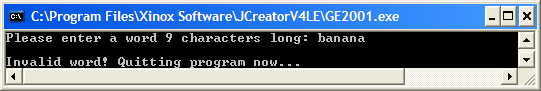
Run 6:



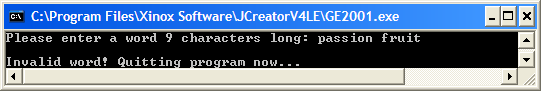
**Exercise 6**

Write a Java program that asks the user to enter a word containing exactly 9 characters. If the number of characters in the word entered by the user is not 9, then put out a warning message and terminate the program immediately. If it is 9, then check to see whether the word entered constitutes a **palindrome**. This is a word that spells the same backwards and forwards. Use the **charAt**() method for this part of the problem. Your program should operate as indicated in the following sample screenshots:

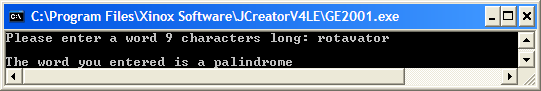
Run 1:



Run 2:



Run 3:



Run 4:

