**Institute of Technology Tralee**

**Computing Department**

**Introduction to Programming**

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

**Practical 2 – Variables & Output Formatting**

After getting through last week's practical you have already done a lot. You are now ready for more Java programming! Many of the tasks you will cover this week such as saving your programs, opening existing programs etc. will be a **repeat** of what you did last week. If you need any help along the way just ask.

**Getting Started ….**

If it is not on, switch on your computer and then when it has booted, log on to the college network using your personal username and password. Then you should launch JCreator as follows:

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

Select **All Programs**->**Java**->**JCreator LE**->**JCreator LE 5.00**

**Using Variables in Programs**

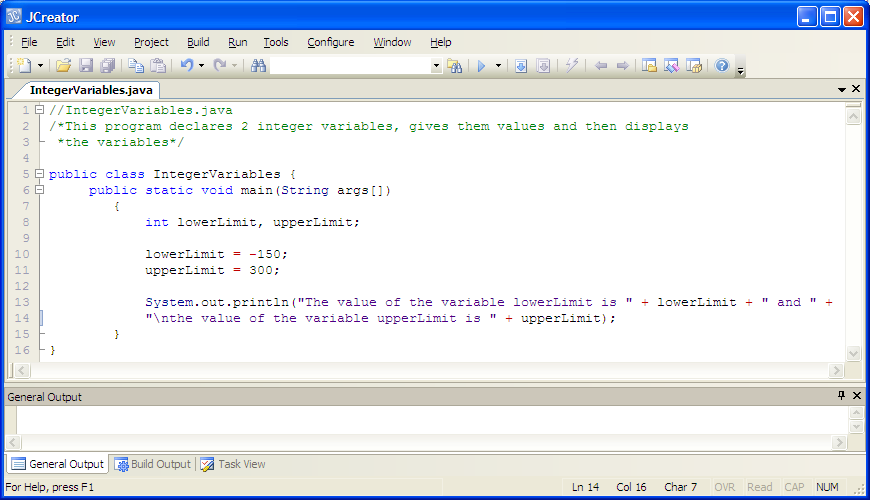
**Variables** are vital in computer programming. A variable is a place for **storing data** during a programs execution. When a variable is defined, it then takes up a certain amount of main memory (RAM) within the computer to store its value. As its title suggests, the **value of a variable can change** during a programs execution.

**Using Integer Variables in Programs**

An **integer variable** is one that can hold a positive or negative **whole number**. Examples of **where integer variables are used** include the following: to hold a person’s age, to hold a temperature value, to hold an hour, minute or second value, the store the number of horses which are running in a race etc.

**Aim**: The purpose of this program is to output the values of two integer variables to the screen

**Java Code**:



**Program Analysis**:

• The program begins with a single-line comment for the programs name and a multi-line comment to explain the purpose of the program.

• After this, there is a class definition for a class called IntegerVariables.

• Next there is a main() method definition.

• the next line of code constitutes **variable declaration statements**. The variables being declared here are integer variables called lowerLimit and upperLimit respectively.

In Java, **variables must be declared before they are used anywhere else in a program**. To violate this rule would constitute a **syntax error**.

When a variable is declared there are 2 parts to the declaration. The **variable must be given a type and a name**.

Here the type is **int** (for integer) and the variable names are lowerLimit and upperLimit.

Note how it is possible to declare more than one variable **of the same type** in the same statement, by simply **separating the variable names with commas**.

The following code would also be valid for the declarations here:

int lowerLimit;

int upperLimit;

but is, of course, less efficient than the version used above.

**Java places a limit on the size of an int variable** and they **can only store values in the approximate range -2 Billion to +2 Billion**.

• the next 2 lines of code set up the variables with the values -150 and 300 respectively. In order to assign values to variables in Java, the **assignment operator** ( = ) is used.

• the last line of code makes a call to the **println()** method which, as always, **displays a message to the output window**. Note the syntax of the println() call in this particular case:

System.out.println("The value of the variable lowerLimit is " + lowerLimit + " and " +

"\nthe value of the variable upperLimit is " + upperLimit);

It begins with the method call itself, then a piece of text (enclosed in quotes as always). Then there is a **+** and then the variable lowerLimit, followed by another + and then some more text. Then there is the + operator to join the next line to the current one, then a \n for a new line, then more text, then a final + operator, then the variable upperLimit. The + is **essential when mixing variables and text** in a println(). It sounds complicated but its easy when you break it all up!

**Organising your Work**

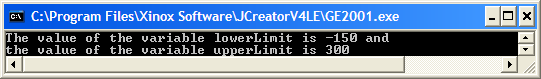
You should have a folder under X: called JavaStuff created already from last time and it should already have a folder in it called Lab1. This time, create a folder called **Lab2** within JavaStuff to save your work from this lab session.

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

At this stage everything is set up for you to write your first Java program which uses variables. Click the **New File** icon on the JCreator IDE and save the file as **IntegerVariables.java** in your Lab2 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 above, 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. In this case it will output the values of two integer variables to the screen as follows:



**Case-sensitivity in Java**

Note that **Java is case-sensitive in terms of variable names**

So if you defined a variable called lowerLimit, as we did in the last program and later on you referred to the variable accidentally as LowerLimit e.g.

System.out.println("The value of the variable lowerLimit is " + LowerLimit + " and " +

"\nthe value of the variable upperLimit is " + upperLimit);

then the program would give a **syntax error**.

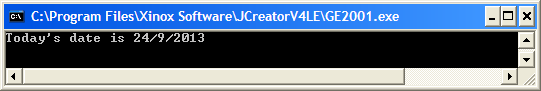
Modify your working **IntegerVariables.java** program now so that the **println()** method refers to LowerLimit rather than lowerLimit and recompile.

You get the message “**Cannot find symbol variable LowerLimit**”. The compiler is complaining that there is something in your program that it does not recognise (LowerLimit) because you must not have declared such a variable. This **error message occurs very frequently** in Java, especially when you start off, since typos are inevitable.

**Java is also case-sensitive in terms of class names and method names.**

**Exercise 1**

Create your own Java program now called **Exercise1.java** which declares 3 integer variables called **day**, **month** and **year** and gives them appropriate values for today’s date. The program should then output the values of day, month and year as indicated in the sample screenshot below, using the 3 variables:



**Floating-point Variables in Java**

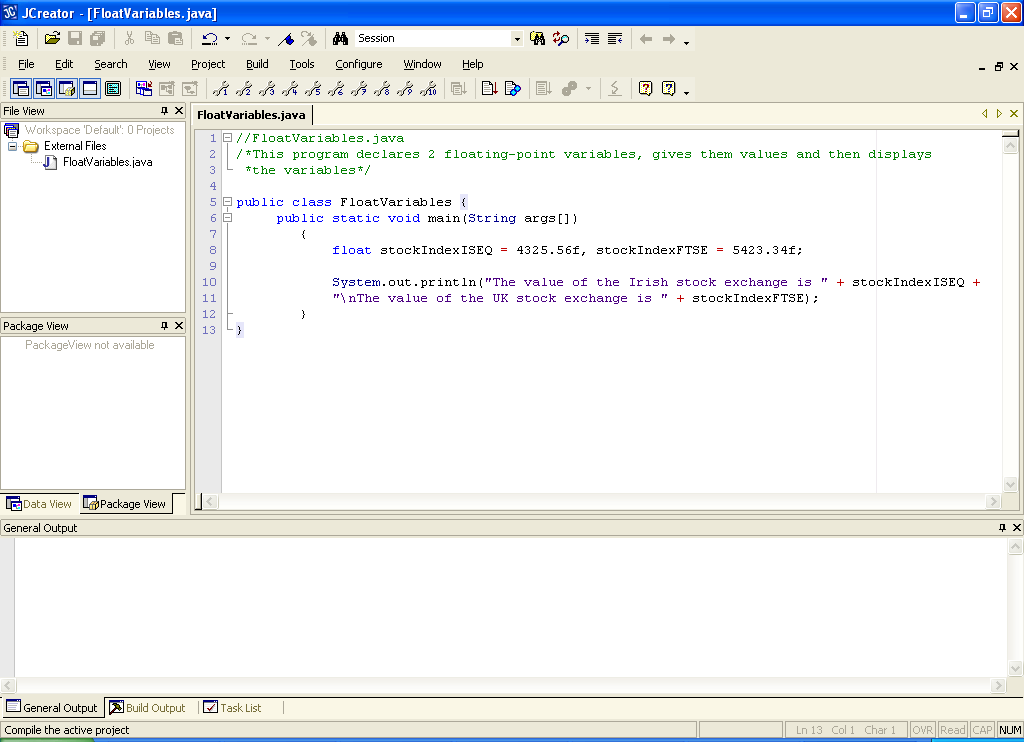
The last program, **IntegerVariables.java** used numeric variables in the form of integers, which are positive or negative whole numbers. However, many programs require variables that are capable of storing **numbers that have a** **fractional part** such as 23.456 or -456.1762 etc. An integer variable cannot be used in these cases so **floating-point variables** are used instead. Floating-point variables can store **much larger numbers** than integer variables also.

A **floating-point variable** is one that can hold a number containing a fractional part. Examples of **where floating-point variables are used** include the following: to hold a person’s weekly wage, to hold a person’s exact height, to hold the fastest time recorded in a 100m sprint race, to hold a stock-market index value etc.

**Using Floating-Point Variables in Java Programs**

**Aim**: To output the values of 2 floating-point variables to the screen

**Java Code**:



**Program Analysis**:

• The program begins with 2 **comments** as normal

• A class definition follows and then the main() method definition as always.

• Line 8 **initialises** two floating-point variables called stockIndexISEQ and stockIndexFTSE with values of 4325.56f and 5423.34f respectively.

**Initialisation** of a variable means **declaring it and giving it an initial value in the same statement**. Recall in IntegerVariables.java how we separated these steps.

Naturally it is **more efficient** **to initialise** variables wherever possible.

Note also in line 8 that the 2 variables have been **initialised together** in the same statement, separated with a comma.

One thing that looks quite odd here is that the numbers are 4325.56**f** and 5423.34**f**. What is the ‘f’ for here? In Java, this ‘f’ is essential in order to avoid a certain kind of syntax error. This comes about because, in Java, there are actually **2 floating-point types** – **float**, which we have just used here is the “single-precision floating-point” type, and **double**, which means “double-precision floating-point”.

double is capable of storing values to a **much greater degree of** **accuracy** than float can and can also store a **much wider range of values** than float. We will use double at certain points in this module also but, in many cases, a float variable suffices for us because the number we are dealing with is relatively small and does not require huge accuracy. It is **more efficient** to use float rather than double wherever possible as it takes up less memory. The way the Java compiler distinguishes between float constants and double constants is by having the ‘f’ tagged onto the end of a float constant, as we have done here.

• Finally, the **println()** method displays the stock index values to the screen and the format here is exactly the same as in the integer variable case, with a **\n** used to put the values on separate lines on the output window.

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

Type in the code for the program above, compile it, debug it and run it. Save your program as **FloatVariables.java** in your **lab2** folder.

**Naming Identifiers in Java**

Collectively, variable names, class names and method names are referred to as **identifiers** (since they allow us to identify certain things within a program). There are a number of general rules which must be followed if you are naming **identifiers** in Java as follows:

(a) Identifiers **must begin with a letter or underscore**

(b) Identifiers **can only contain letters, digits, underscore or currency symbols**

(c) Identifiers **cannot contain a blank**

(d) Identifiers **cannot be keywords**

So, armed with the knowledge above, which of the following are valid Java identifiers?

(I) public (II) 2ndNumber (III) maximum height (IV) age (V) first-name

(VI) day\_of \_week

It is essential that **variable names should be meaningful** also - to **improve the readability** of a program e.g. you should declare an integer variable called **hours** or **hrs** to store a certain number of hours rather than **h** which is quite vague and not very obvious to somebody reading your code.

You should **always think about making your code as easy as possible for others to read**.

What would be a good meaningful variable name for storing how tall somebody is?

What would be a good meaningful variable name for storing the weight of the person with the lowest weight in a group of people?

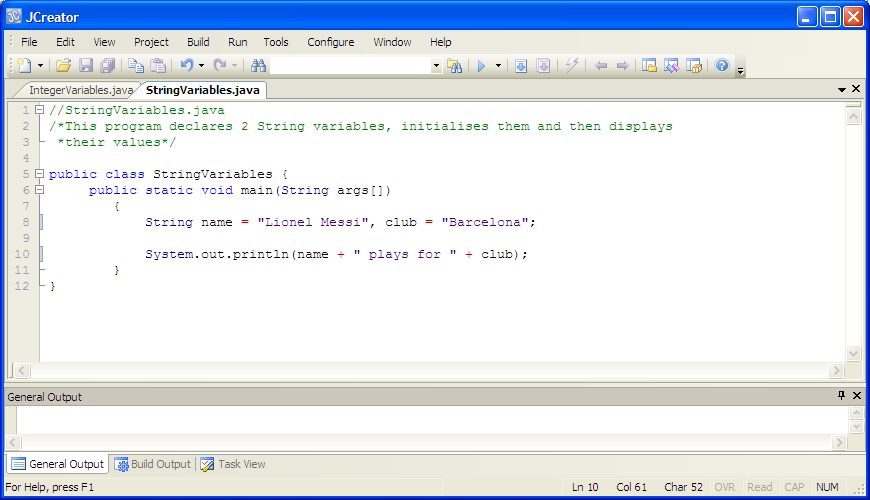
What would be a good meaningful variable name for storing the average age of a set of students?

**Using String Variables in Java Programs**

A **string variable** is used to store any sequence of characters you like such as letters, numbers and symbols. Strings may be used to store someone’s name, address, PPS number etc.

**Aim**: To output the values of some string variables to the screen

**Java Code**:



**Program Analysis**:

• Line 8 **initialises** 2 String variables to the values indicated. Note that String is spelt with an uppercase ‘S’. This is because **String is actually a class** in Java, whereas int ,float and double are called **primitive data types**.

Note that String variables are assigned pieces of text - **String constants** – which are always **enclosed in double-quotes**, just like the text messages in the println(). In fact the text messages in a println() are also String constants. It would constitute a **syntax error** to omit the double-quotes when assigning a value to a String variable.

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

As usual, type in the above program, compile, debug and run it. Save it as **StringVariables.java**. This may seem tedious but **programming is all about practice** – the more time you spend typing in code, the quicker you will get to grips with the syntax of the language and the more errors you will encounter (and hopefully fix) along the way.

**Exercise 2**

Write a program called **Exercise2.java** that uses variables of the appropriate type for storing the following information for a book. Try to **use numeric variables wherever you can**:

Title – Harry Potter and the Prisoner of Azkaban

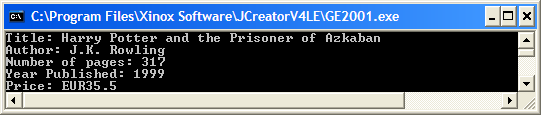
Author – J.K. Rowling

Pages – 317

Year of Publication – 1999

Price – 35.50

The program should then display this information to the screen exactly as indicated in the screenshot below, using the variables created earlier. Try to use just one **println**() call here when displaying the information:



Notice in my case above that the price doesn’t appear as 35.50, the trailing zero has been automatically wiped from the floating-point variable’s value. Don’t worry about that for now though, we will rectify it shortly when we cover **formatting**.

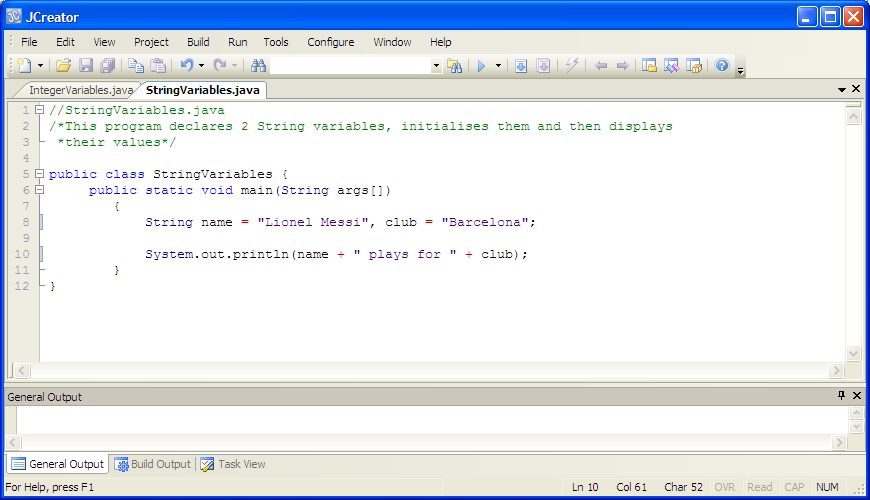
**Conventions in Java**

You may have noticed that whenever I have created variable names, they always **began with a lowercase letter**. Also, whenever I had a double-barreled variable name such as stockIndexISEQ the variable began with a lowercase letter but the first letter in the second word began with a capital letter as did the first letter in the 3rd word (these are initials so they all appear in uppercase). If I had a variable name containing even more words, each first letter would be a capital. However, doing this is only **convention** and the **compiler will not signal a syntax error should you break this convention**.

In programming, if something is a **convention** then it means that the vast majority of programmers do it but it is not a grammatical rule of the programming language. It is good to abide by conventions though so that our programs look “professional”.

Method names follow the same conventions as variable names but **class names**, by convention, always **begin with an** **uppercase letter** and each subsequent word in the class name is also uppercase e.g. the classes StringVariables, FloatVariables, String, System etc.

Another convention involves the use of **whitespace characters** (blank spaces, blank lines and indentation). Recall the following program we just had:



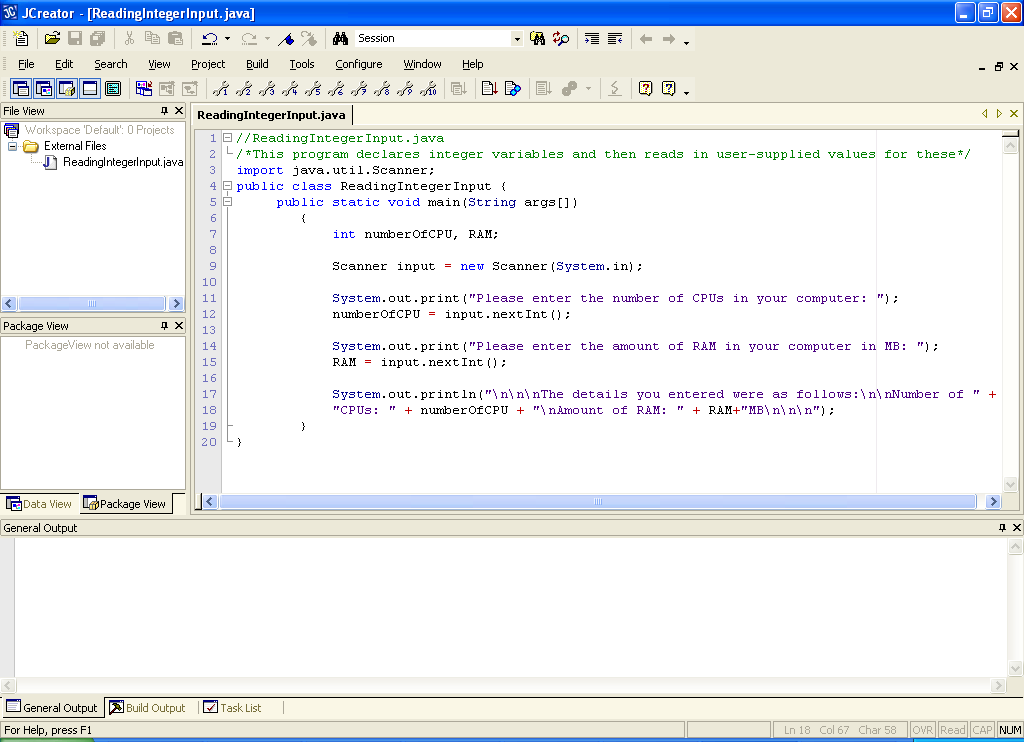
Notice that I have put a **blank line** in between the multi-line comment and the class definition. Also I have put a blank line between the variable initialisation section and the println() statement. These are conventions that are widely adopted but again are not mandatory. The **aim here is to make the program more readable** and less cluttered though the use of whitespace. Note also the use of **indentation** throughout. Again, this is merely convention and is simply meant to make the program look more visually appealing.

**Reading in Values Supplied by the User**

All the programs you have seen so far in this module have involved outputting information to the screen - textual information and integer, floating-point and string variable information, using "**hard-coded**" constant values. Now we want to move on to programs that are capable of **reading information in from the user as a program executes**, which is essential in almost all real-life software systems.

**Aim**: The purpose of this program is to read in values for two integer variables from the user and then output the values entered to the screen

**Java Code**:



**Program Analysis**:

• After the comment there is an **import** statement. In Java, predefined classes belong to **packages** and, in this case, we need to use a predefined class called **Scanner** from a package called **java.util**. Before we can use the class, we must import it into our program (there is one exception to this rule which we may discuss later). The **Scanner** class allows us to **read information from the keyboard** (as well as other sources) by using its appropriate methods.

When an import needs to be carried out in a program it **must always come before the class definition**. My own personal style is to place import statements after any opening comments but many programmers put them before the opening comments.

• Within main() two integer variables are declared and notice that they are given no initial values – this makes sense as we are awaiting the user to supply the values. We could, however, supply initial values to the variables if we wished as they would end up being overwritten by the user-supplied ones anyhow.

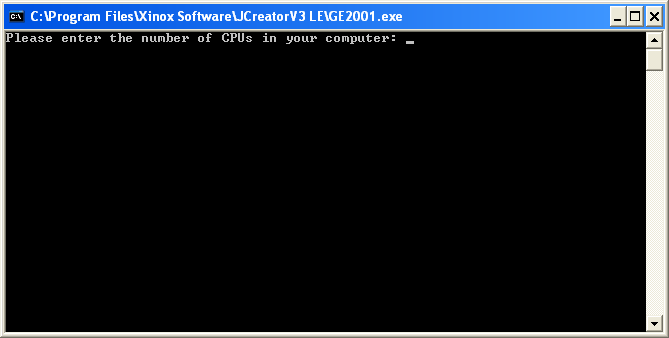
• The next line of code looks a bit unusual.

Scanner input = new Scanner(System.in);

Here we are **creating an object of type Scanner** using the **new** keyword**.** Objects are the basic building blocks in OO languages such as Java so it is impossible to avoid creating objects for very long if we want to write useful programs! The object we have created here will be manipulated in our program by the **object reference** called **input** and the object will be associated with the keyboard, which is what **System.in** refers to here. You may recall from the last practical that System.out refers to the output window, so it makes some sense hopefully that System.in refers to the keyboard (an **in**put device). So, in our program the object reference called input is essentially a reference to the keyboard, which is where the user-supplied values will be read from later.

• Line 11 contains a prompt to the user to supply information. Notice that the method used here is print() rather than println(). These methods are identical in their functionality except for the fact that println() also places the cursor on the beginning of the next line when it has displayed its message. When prompting the user for information, it looks better to keep the cursor flashing at the end of the prompt message – this is achieved by using print() instead of println(). This is purely cosmetic however and the program would still function perfectly with println().

• On line 12, the input object reference created earlier is now being put to use. The method **nextInt**() is being called on the object reference and this has the effect of firstly **halting the program’s execution** in order to give the user the chance to enter some data – this is the effect when the program executes:



and then, when the user has entered their data and hit return (to signal the end of data input), it reads the number entered by the user and assigns it to the variable numberOfCPU.

• Lines 14 and 15 work in an identical manner to lines 11 and 12.

• The program finishes off by displaying the values stored in the variables.

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

You are now ready to write your first Java program which deals with reading in user-supplied values into variables. Type in the code from the program above into a fresh window, compile, debug and run as usual. Save the program as **ReadingIntegerInput.java** and save it in your **Lab2** folder.

**Reading User-Supplied Values into Floating-Point and String Variables**

As you might have guessed, there is very little **difference when reading values into either floating-point or String variables**. The only difference is that instead of using the nextInt() method as we did here we would use:

**nextFloat**() to read in a floating-point value,

**nextDouble**() to read in a double-precision floating-point value

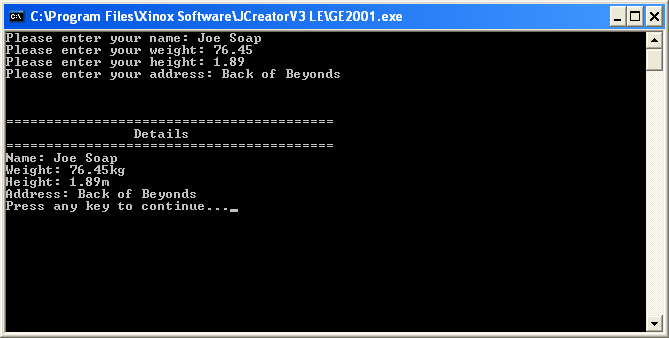
**nextLine**() to read in a string value (need to be careful using this one though – more on that below)

**Exercise 3**

Write a program called **Exercise3.java** that asks the user to enter their name, weight, height and address. The program should then display the values entered exactly as indicated in the screenshot below.

Be extra careful when using the nextLine() method here for the **address**. The problem is that nextFloat() (and nextInt() and nextDouble()) does not actually read the end-of-line character ‘\n’ when the user hits return after entering data and when nextLine() is called to read in the address, it will immediately read the remainder of that line of input i.e. it will read the ‘\n’ and **return the empty string for address**, which is not what we want of course. To avoid this situation is easy though, just **put in an extra input.nextLine() call** when dealing with the address, the first one will get rid of the ‘\n’ and then the second one will read in the actual address you have typed.

You should aim to **write your program in the most efficient way you can** – use a **single println()** for the “Details” section of the output and as usual **include comments** and **make your variable names meaningful**.

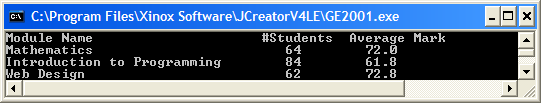


**Formatting Output in Java**

There is a general requirement in software systems to be able to present information in a user-specified format. For example, imagine that we have the following raw data given to us in a table:

|  |  |  |
| --- | --- | --- |
| Module Name | # Students Enrolled | Average Mark |
| Mathematics | 64 | 72.031 |
| Introduction to Programming | 84 | 61.845 |
| Web Design | 62 | 72.774 |

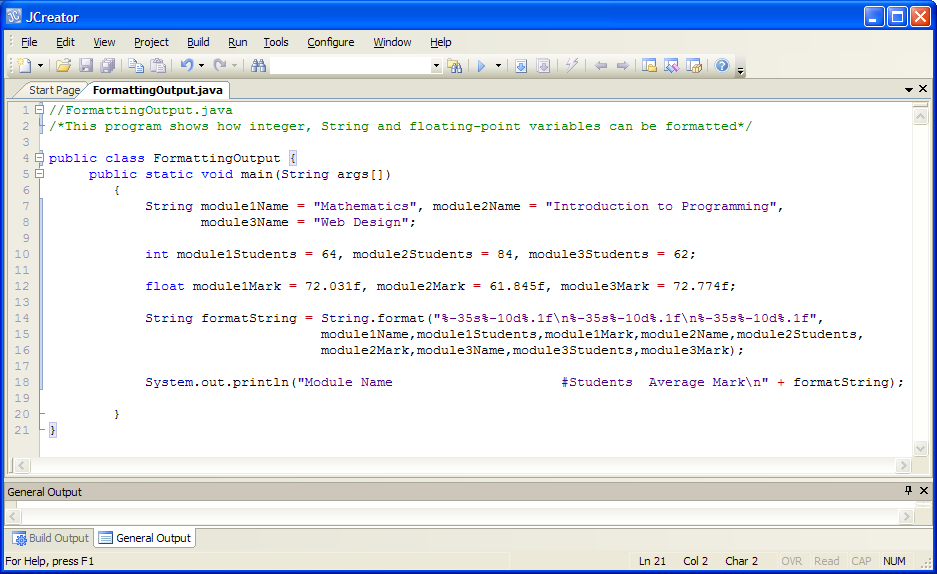
We now wish to format this data in our own Java program, to present it in a nice neat tabular fashion as follows:



Notice how the columns line up nicely and also that the average marks have been displayed correct to 1 decimal place.

**Aim**: To format the output to display a set of values in a neat tabular fashion

**Java Code**:



**Program Analysis**:

• Line 7/8 **initialises** 3 String variables with the module names, line 10 initialises 3 integer variables with the number of students enrolled for each module and line 12 initialises 3 float variables with the average mark for each module.

• In lines 14-16 we have the all important “format string”. There is a call to a method **format**() which does the job of allowing us to dictate how we want to format the information. The format string itself here is the part

"%-35s%-10d%.1f\n%-35s%-10d%.1f\n%-35s%-10d%.1f"

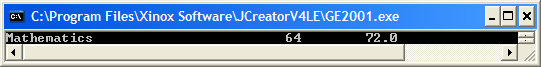
This looks terrible at first glance but it can be broken down into a number of simpler parts. What we want to do in this case is to display the output so that it appears in a neat tabular fashion.

The very first piece of information to be displayed is the value of the String variable module1Name. This variable stores the value “Mathematics”. As this is a String variable we dictate its format using the **%s** **format specifier**. On its own the %s format specifier won’t do anything useful for us, so we need to add extra bits of information to state exactly what we want to do. In this case, I want to allow a “**width**” of 35 spaces on the output window to display the value of this String variable. This number means that the word “Mathematics” will display within a field that is 35 characters wide and the remainder of the field will be automatically padded with blanks. I also want to display the word “Mathematics” in a **left-justified** manner within this 35-wide field. In order to achieve this I place the – character before the number 35. If I were to omit the – character then the word would be displayed in a right-justified manner within the field instead.

The next piece of information to be displayed is the value of the integer variable module1Students. This variable stores the value 64. As this is an int variable, we dictate its format using the **%d format specifier**. Again, this format specifier won’t do much for us on its own so we add extra bits of information to state exactly what we want to do. Here, I want to allow a “width” of 10 spaces on the output window to display the value of the int variable. Also, I want to left-justify the display of the value, so I add the – character before the number 10.

The next piece of information to be displayed here is the value of the float variable module1Mark. This variable stores the value 72.031. As this is a float variable, we dictate its format using the **%f format specifier**. In order for it to be useful, we must add extra bits to it in order to state exactly what we want to do. As this is the last (rightmost) column, I don’t need to specify a field-width (I could do so but it’s not necessary). However, I do want to display the marks correct to **1 decimal place** (the raw data in the table has them to 3 decimal places), so I do this by putting .1 before the f in the format specifier. If I had wanted to display correct to 2 decimal places, it would have been .2 etc.

So at this stage we have effectively created the format string for the first 3 pieces of information, which would display as follows

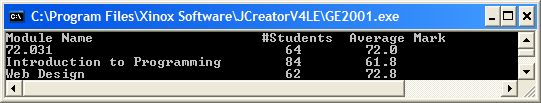


|--------------- 35 wide -----------------|- 10 wide -|--remaining space--|

After this you notice in the format string that there is a \n character. This is to ensure that the next chunk of output displays on the next line of the output window. Without it, the information would continue to display on the same line.

The good news is that the rest of the format string is just a repeat of this pattern because we effectively wish to format each set of 3 chunks of data in exactly the same manner.

Notice that in the call to format() there are 2 parts, the first is the “format string” that we have been discussing for the past while and the second part is the list of comma-separated variables whose values we wish to format. The order of the variables must match the order of the format specifiers so e.g. it would cause a **logical error** if we accidentally tried to display module1Mark first using the %-35s format specifier. The data would display as



The call to format() returns the format string which is then stored in the String variable formatString.

• In line 18 we display the table headings followed by the format string.

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

You are now ready to write your first Java program which deals with formatting output. Type in the code from the program above into a fresh window, compile, debug and run as usual. Save the program as **FormattingOutput.java** and save it in your **Lab2** folder.

Note that if you are viewing the output from the program above in the “General Output” window, the data might be appearing out of line. If so, it is because the default font used for the window is not “monospaced” – in a monospaced font every character takes up exactly the same amount of width in the window. You can go to **Configure->Options->Workspace->Output View->Font** and choose the “Courier” font if you want to use a monospaced font to see the output inline.

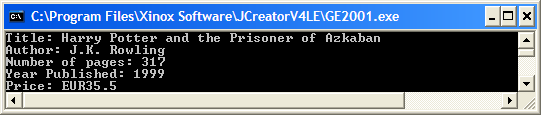
**More on Format Specifiers**

We have really only scratched the surface on formatting output here but it is plenty for now. The most common use we will have for format specifiers in this module will be to use them to correct a result to a certain number of decimal places.

One important feature of format specifiers is that they are **case sensitive** so e.g. you must use %d rather than %D (which would generate a **runtime error**).

**Formatting an Earlier Exercise**

Recall from earlier that when you ran Exercise2.java it displayed as follows:



As you can see, the information is out of line and not very neat. You should now open the **Exercise2.java** program, save it as **Exercise2Formatted.java** and **modify** it using your knowledge of format specifiers so that the output displays as follows instead:

