**Institute of Technology Tralee**

**Computing Department**

**Introduction to Programming**

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

**Practical 3 – Arithmetic and Calculations**

At this stage we have introduced keyboard input and screen output and a useful method called format() for formatting the display. This lab sheet will cover the Java arithmetic operators and show how they can be used to perform various calculations and conversions.

**The Java Arithmetic Operators**

In Java, arithmetic calculations may be performed through the use of 5 operators as follows:

+ addition (sum)

- subtraction (difference)

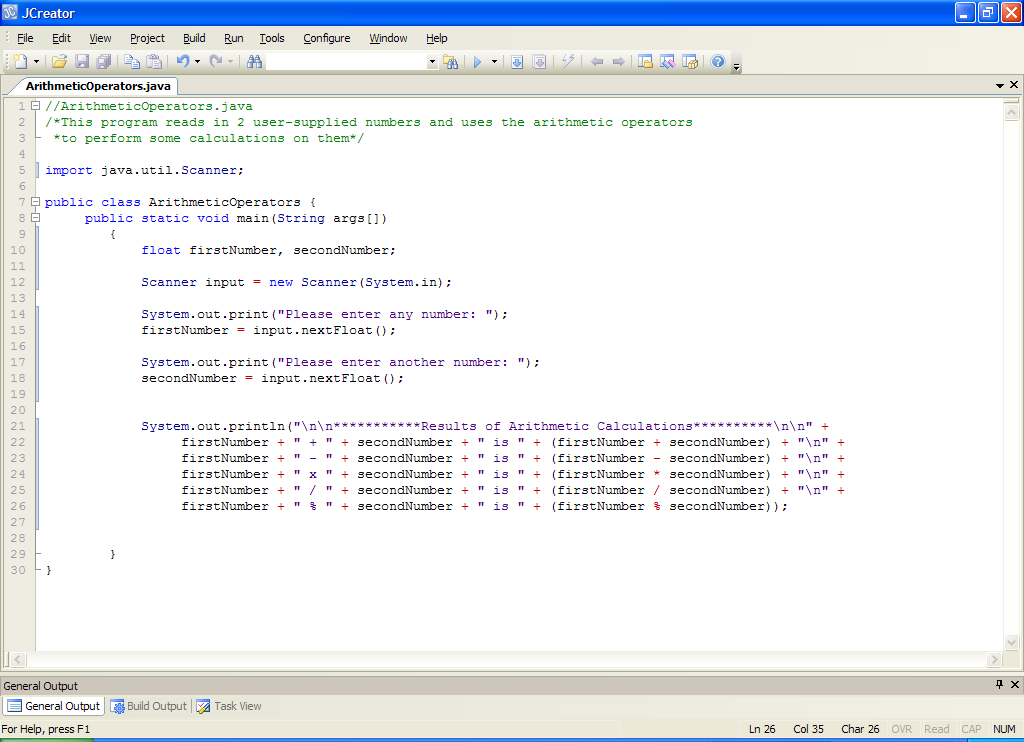
\* multiplication (product)

/ division (quotient)

% modulus (remainder)

**Aim**: The purpose of this program is to read in any two user supplied numbers and compute various results based on them using the arithmetic operators

**Java Code**:



**Analysis of program**

• After the comments comes the **import statement** for Scanner.

• Two floating-point variables are then declared to store the 2 numbers that the user will enter later.

•Next we have the line of code that creates the Scanner object which enables us to read information from the keyboard (**System.in**).

• Next the user is prompted for the numbers and each is read in using the **nextFloat**() method.

• Lines 21-26 contain the actual calculations. Note that it is perfectly **acceptable to place calculations directly into a println() method call** as shown. It is **more efficient to place calculations directly within an output method** rather than using additional variables for storing the calculation results e.g. we could have done the following instead for adding the numbers:

**System.out.print("Please enter any number: ");**

**firstNumber = input.nextFloat();**

**System.out.print("Please enter another number: ");**

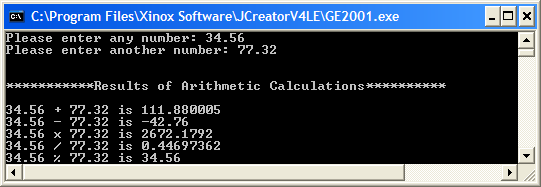
**secondNumber = input.nextFloat();**

**sum = firstNumber + secondNumber;**

but this **would have introduced an extra variable** called sum which is essentially unnecessary in this case. Likewise, we have saved ourselves the trouble of having difference, product, quotient and modulus variables by including the calculations directly within the println(). Remember to always think about **efficiency**.

The results of the calculations will then be displayed directly to the output window.

A sample run of the program gives the following here:



Note how the **modulus** (%) operator works. It determines the **remainder upon division** of one number by another e.g.

**10 % 3** gives 1 because when you divide 10 by 3, it goes into 10 three times evenly but there is a remainder of 1.

**15 % 5** gives 0 because when you divide 15 by 5, it goes into 15 three times evenly and there is no remainder i.e. a remainder of 0.

So what is 17 % 4?

So what is 98 % 13?

Believe it or not, the % operator can be used within programs surprisingly often and you will see it a few times in this module.

Note that when floating-point calculations are performed in Java, the number displayed will contain at most 9 significant digits in total, including those after the decimal point. If further accuracy were needed, we could take advantage of the format() method used in the last practical sheet e.g. to display the quotient, firstNumber/secondNumber, to 10 decimal places, we could do the following:

System.out.println(firstNumber + " / " + secondNumber + " is " +

String.format("%.10f",(firstNumber / secondNumber)));

**Issue with Floating-point Arithmetic in Java**

The above sample run gives the correct answers for 4 out of the 5 calculations. However, the first one is actually logically incorrect! The answer should be 111.88 exactly! The problem here is simply that Java’s floating-point arithmetic is based on a certain standard (called IEEE 754) which is, like all floating-point arithmetic, limited by the hardware within the computer itself (more in your computer architecture course in semester 2). Floating-point arithmetic is carried out in binary 1’s and 0’s and there are a limited number of registers available to carry out calculations (normally 32 or 64 bits nowadays). Therefore, certain numbers cannot be stored accurately and **roundoff** occurs, hence the **logical error** above.

There is a very good web page on this topic at <http://mindprod.com/jgloss/floatingpoint.html>

which is worth a quick look if you get the chance.

Such small errors are not a big deal for us but they could be catastrophic to other developers. These people use the primitive data types int and long (which can be stored exactly in binary form) as well as special Java classes such as BigInteger and BigDecimal to carry out their calculations, since accuracy is absolutely guaranteed with them. We might get a chance to look at BigDecimal in semester 2 if we get the chance as it is quite easy to use – for now though, we will live with these small losses in precision.

**Creating a Folder to Store This Weeks Work**

Create a new folder within the JavaStuff folder on your X: drive called **Lab3** to store this weeks work.

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

Now you should type in the program above and save it as **ArithmeticOperators.java** in the Lab3 folder. You should then compile it and run it a few times with different values.

If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier. If you spot any differences correct them and compile again until the program is syntax error-free.

You can use your calculator (or the calculator on the computer) to see how close Java’s results are to the calculators.

**Formatting the Previous Program**

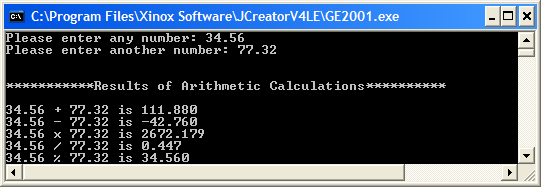
To get some more practice with output formatting, take the ArithmeticOperators.java program that you have been working on and save it as **ArithmeticOperatorsWithFormatting.java**. Now you should modify the existing code so that the results of all the calculations get displayed correct to **3 decimal places**. You can use the code snippet discussed earlier i.e.

System.out.println(firstNumber + " / " + secondNumber + " is " +

String.format("%.10f",(firstNumber / secondNumber)));

as a guide when making your modifications here.

Given the same input values as before, the program would now run as follows:



**Division of 2 Integers in Java**

One thing to watch out for in Java arithmetic is that the **division of one integer by another actually results in an integer**. If there is a decimal part, it will be “chopped off” completely e.g.

12/5 gives 2 and not 2.4

-8/3 gives -2 and not -2.6666666….

This can actually be very useful when programming at times and we will take advantage of it in certain types of conversion problems.

**Operator Precedence**

All programming languages contain numerous **operators** that carry out specific jobs. You already know that there are 5 arithmetic operators in Java and their **precedence refers to the order in which the compiler actually evaluates them**. The compiler effectively views certain operators as being “more important” than others and this dictates how they are evaluated at runtime.

In Java, in terms of the 5 arithmetic operators, their precedence from highest to lowest are as follows:

\* / % highest

+ - lowest

In cases of **equal precedence**, the operators are **evaluated in a left to right fashion** as they are encountered.

So if the compiler sees the following expression in a program:

**x = a \* b % c – 4 + 6**

it would evaluate it as follows:

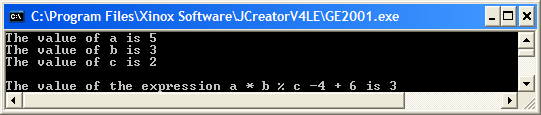
1. It would calculate a\*b
2. It would then get the reminder upon division of this result by c
3. It would then subtract 4 from the result in part 2
4. It would then add 6 to the result from part 3
5. It would then store the final result in the variable x

So, for example, if a has value 5, b has value 3 and c has value 2, then the following will happen:

1. a\*b becomes 5\*3 = 15
2. (a \* b) % c becomes 15 % 2 = 1
3. (a \* b % c) – 4 becomes 1 - 4 = -3
4. (a \* b % c – 4) + 6 becomes -3 + 6 = 3
5. finally x is assigned the value 3

**Exercise 1**

You should now write a program called **Exercise1.java** that tests out the above expression to prove to yourself that x does evaluate to 3. You **should create variables** a, b, c and x in your program and **initialise** a, b and c to the values 5, 3 and 2 respectively – note that there is **no input from the user required at all** here. You then set x to the value of the expression above and display the result. Your program should run exactly as indicated in the screenshot below:



**Overriding the Rules of Operator Precedence**

Consider the following expression



This type of expression can sometimes lead to logical errors in a program because the result may not be what the programmer expected as a result of the rules of operator precedence. Imagine that the programmer wrote the following code intending to calculate the expression above:

result = a \* b / c \* d;

In this case a **logical error** exists in the programmer’s code because the r**ules of operator precedence** here will actually evaluate the expression as



To overcome such errors is very straightforward, you simply **use parentheses** in your expression to indicate the order in which you wish things to be evaluated. So, in this case the programmer’s code should really be

result = a \* b /( c \* d);

In this case the multiplication of c by d will occur first, then a will be multiplied by b and finally the division will occur and the logically correct result will be evaluated.

**Constants in Java**

Recall that a variable is a place for us to store data as our program executes. A variable can change as the program executes.

In Java, it is also possible to define a **constant** in a program. A constant, as its name suggests, **cannot change** during a program’s execution. Once you define a constant, its value can never change and any attempt to change it will cause a **syntax error**.

Constants are formed by using the **final** keyword when the variable is being declared e.g.

**final int MAXIMUM\_ATTEMPTS = 3;**

this defines an integer constant called MAXIMUM\_ATTEMPTS and sets its value to 3.

Constants are useful as they can help make a program **more easy to understand**. In the case above, the constant might form part of a game where the user is given a maximum of 3 attempts to carry out a certain task. If the constant wasn’t used the game could still work by just “hard-coding” the number 3 into the games logic where required. However, anyone reading the code would find it much more difficult to understand where the “magic number” 3 comes from and what it is meant to represent.

Note that, by **convention**, constants in Java are always in UPPERCASE with multi-word constants separated by underscores.

Of course, it is possible to define float, double and String constants also.

You will get a chance to use a constant in the next exercise.

**Exercise 2**

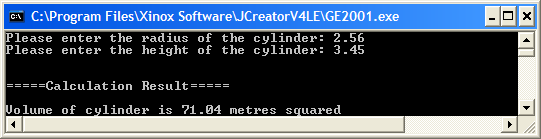
The volume of a cylinder can be determined by using the formula



where you can assume π is a **constant** that has the value 3.142, r is the radius of the cylinder and h is its height. You can define π using the name PI.

Write a program called **Exercise2.java** that prompts for and reads in the value for the radius and height of a cylinder and uses it to determine its volume. The volume should then be displayed correct to **2 decimal places** via the **format()** method.

The program should give you **exactly** the following output when it runs. You can use the values indicated when testing your program:

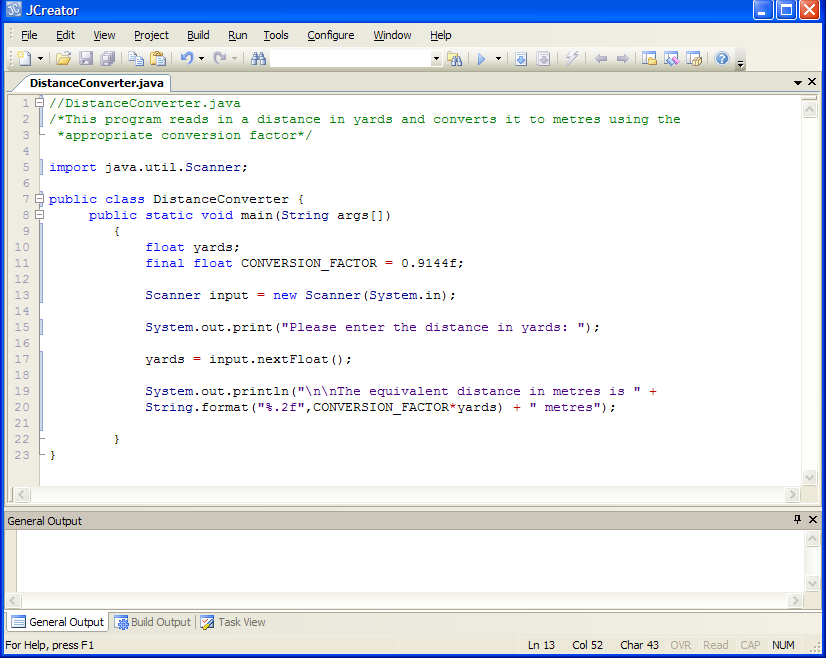


**Java Programs which Perform Conversions**

**Aim**: The purpose of this program is to read in a distance in yards and convert it to metres. The amount in metres should then be displayed to 2 decimal places.

1 yard = 0.9144 metres and this conversion factor should be created as a **constant**.

**Java Code**:



**Program Analysis**:

• After the opening comments is the usual import statement.

• 1 variable and a constant are then created. Note that the constant is written in uppercase with an underscore separating the two words, as per the Java conventions.

• The distance in yards is then prompted for and read in.

• The call to println() displays the result of conversion to 2 decimal places using format(). The calculation is performed inside the method for efficiency.

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

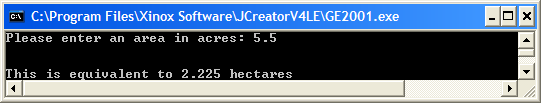
Type in the code for the program above, save it as **DistanceConverter.java** and then compile, debug and run it.

Run this program **a few times** using various values for the yards. Use the **calculator utility** on the computer to make sure the values are correct.

**Exercise 3**

Write a program which asks the user to input an area in acres and converts it to hectares. The program should output the amount in hectares correct to **3 decimal places**. Use the conversion factor **1 acre = 0.4046 hectares.** Save your programas **Exercise3.java**

A sample run of this program would be as follows and your output should emulate this.



**Testing & Categories of Error in Java programs**

**Testing** is a vital part of structured program development since without **complete testing** we cannot know whether the software we have developed actually works or not. A program which runs properly for one test value may not run for another and this can be a big problem.

***Syntax Errors***

The type of error which you have seen most often over the past few weeks have been **syntax errors,** which arise when the syntax of your program disobeys one of the grammatical rules of the programming language.

Frequently occurring examples of syntax error include

● misspelling a variable or method name

● forgetting semi-colons at the end of statements

● forgetting double-quotes around a text message

● forgetting the closing parenthesis when calling a method

● forgetting the closing curly brace on the class definition and/or main() method

● forgetting the f when initialising a float variable

● calling a method incorrectly

The good thing about these errors is that **they can be detected by the compiler** and it is up to us as programmers to look at the errors given upon compilation and correct them and then recompile to see if the problem is solved.

***Logical Errors***

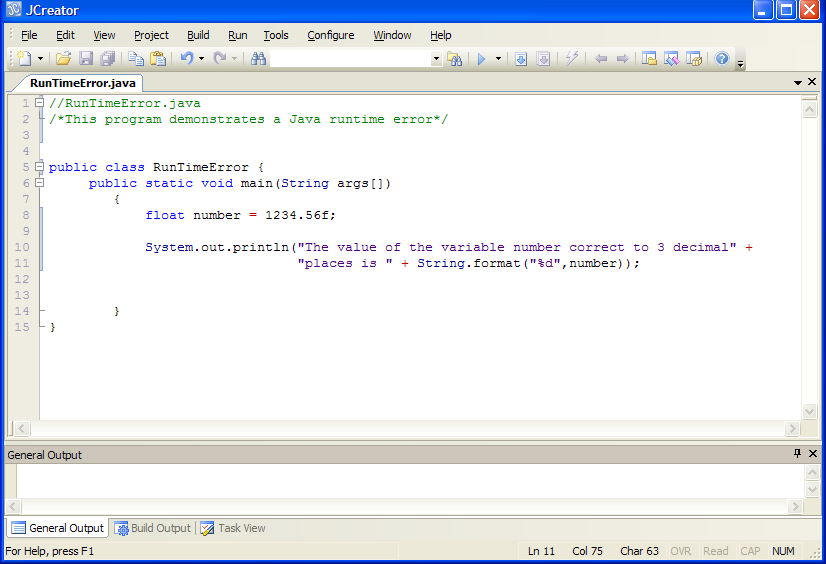
Another type of error, called a **logical error**, commonly occurs in programs and is usually more difficult to fix than the syntax errors since the **computer gives absolutely no warning** **that anything is wrong**. With a logical error, the rules of the language are not broken but the **output is not exactly what you expect it to be**.

A perfect example concerns the **Exercise3.java** program above. If, for example, the conversion factor **0.3046** was used by accident instead of 0.4046, the program would still compile perfectly well and it would also run perfectly well, assuming that a valid number for the acres is supplied. In this case the error is a logical one since the **logic of the program is at fault**. Similarly, if we wished to see the amount in hectares correct to 3 decimal places and used .1f by accident in the **format()** method, this would also be a logical error. Any **misuse of a formula** in a program always constitutes a logical error. You will meet many more examples of logical errors as the course goes on.

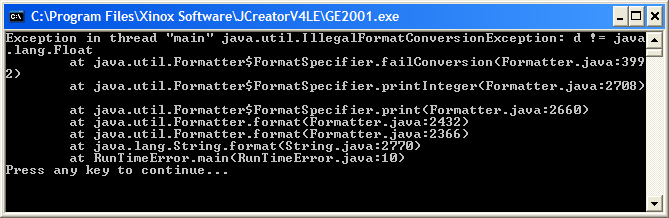
***Runtime Errors***

The final category of error is the **runtime error** which you may not have seen any example of yet. A runtime error only occurs at runtime, when the program is actually executing. Therefore, it **cannot be detected by the compiler**. When a runtime error occurs the **program crashes**, unless the programming language has the means for avoiding such crashes through an **exception-handling mechanism** where the programmer can anticipate and code for such eventualities. Java has such a mechanism but it is **well beyond the scope of our module** here.

An example of a situation where a runtime error occurs is illustrated in the following demo code:



Running the above program above produces the following output here:



The contents of the console window above indicate that an “Exception” has occurred in the main() method. An exception is another term for a runtime error. Effectively the program has crashed and is of no further use.

As a matter of interest, the first line of the output above gives us a clue as to what caused the program to crash here. It indicates that the exception that occurred in this particular case was an “IllegalFormatConversionException” and that “d != java.lang.Float” – it is the last part that really matters in terms of fixing the problem here. The last line above tells us the line at which the crash actually occurred in our program (on line 10 in the call to println()). It is basically complaining that the format specifier %d has been used to try to format a floating-point variable, which is illegal. The problem can be fixed by changing the format specifier from %d to %.3f.

As a programmer you should always try to **test your programs fully** for various kinds of inputs so that you can eliminate runtime errors. It is **only through thorough testing that you can be confident your software is free from logical and runtime errors**.

**Exercise 4**

In kinematics, a branch of physics, the distance, s, traveled by an object is given by the formula

s=ut+at2

where:

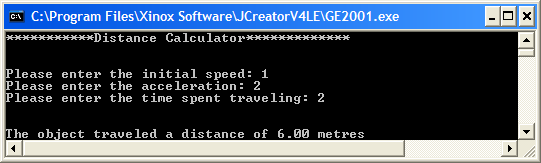
u is the initial speed of the object

t is the time spent traveling

a is the acceleration of the object

Write a program called **Exercise4.java** which prompts for and reads in values for initial speed, time and acceleration and determines the distance traveled by the object. Your program should run as indicated in the following screenshot, with the distance corrected to **2 decimal places**:

[**N.B.** in the formula **s=ut+at2** only the **t is to be squared** and nothing else]



When you have your program running **make sure it is logically correct** by testing it using the following values for a , t and u:

**a=0,t=0,u=0**=>s=0.00m

**a=1,t=1,u=0**=>s=0.50m

**a=1,t=1,u=1**=>s=1.50m

**a=2,t=1,u=2**=>s=3.00m

if you get values other than these for **s** you haven’t got the formula coded right.

**Exercise 5**

The gravitational force, *F*, exerted on one object by another is given by the formula



where you can assume that

***G***is the gravitational constant = **6.67300 × 10-11** m3 kg-1 s-2

***m1***is the mass of the first object

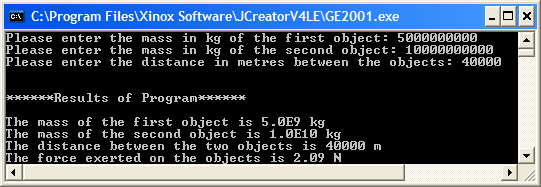
***m2*** is the mass of the second object

***d*** is the distance between the two objects

In your program you should make G a **constant** with the value indicated above. To write this value in your program let G be set to the value **6.67300E-11** (this is called **scientific notation** format).

Write a Java program called **Exercise5.java** that prompts for and reads in the value for the *mass* of the first object, the *mass* of the second object (assume both of these are **floating-point** values) and the *distance* between the two objects (assume this will be a **whole number**) and uses these to determine the *gravitational force* exerted on the objects.

Using the test values as indicated in the screen shots below, the program should give you **exactly** the following output when it runs. Note that the numbers displayed in **scientific notation** form e.g. 5.0E9, will be displayed like this **by default** as they are so large and **you need do nothing special in your code to achieve this effect**.



**Flaw with the Last Program**

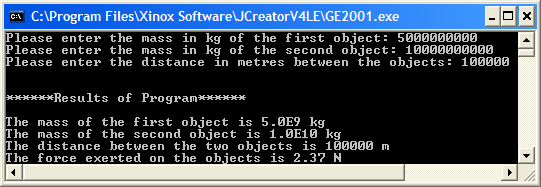
You may not have noticed any problem with the last exercise and hopefully you managed to get it to work. However, proper testing should throw up a problem with the program. The trouble is with the formula. In this case, it was:



Assuming you coded it correctly, it probably looked something like this:

G\*mass1\*mass2 / (distance\*distance)

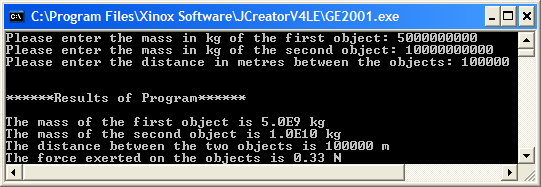
Now the coding here looks spot on and we have even used the parentheses as required (otherwise we would have another logical error). But the flaw remains. The trouble is that in Java, integers can only be so big – approximately 2 Billion is the maximum size of a valid integer (which is relatively small, imagine trying to store Bill Gate’s wealth as a Java integer!). Here there is a calculation that involves multiplying two large integers together (distance\*distance). We get away with it in the sample screen shot above because the value of distance is only 40000, squaring this is less than the 2 Billion limit. However, if we were to enter a larger distance value, say 100000, then we would go beyond the limit and we would get a **logically incorrect answer**. The Java runtime tries to deal with the problem but it can’t, even though the answer we get looks normal enough. However, when you cross-check with a calculator, you quickly discover the problem. The calculator indicates the answer should be 0.33N but we get 2.37N, a **logical error**.



As always, however, there is a solution to this problem. It involves what is called **type-casting**. With type-casting, you **force a variable to become a different type momentarily**. Here, the distance variable started off as an integer and we saw the problem that could happen if squaring that integer produced a number too large. So, if we type-cast the distance variable to become a float variable just while the calculation is taking place we can then get over the relatively low limit of 2 Billion. This is because in Java, **mixed arithmetic results in a floating-point number**. To achieve type-casting here is straightforward – our code becomes:

G\*mass1\*mass2/ ((**float**)distance\*distance)

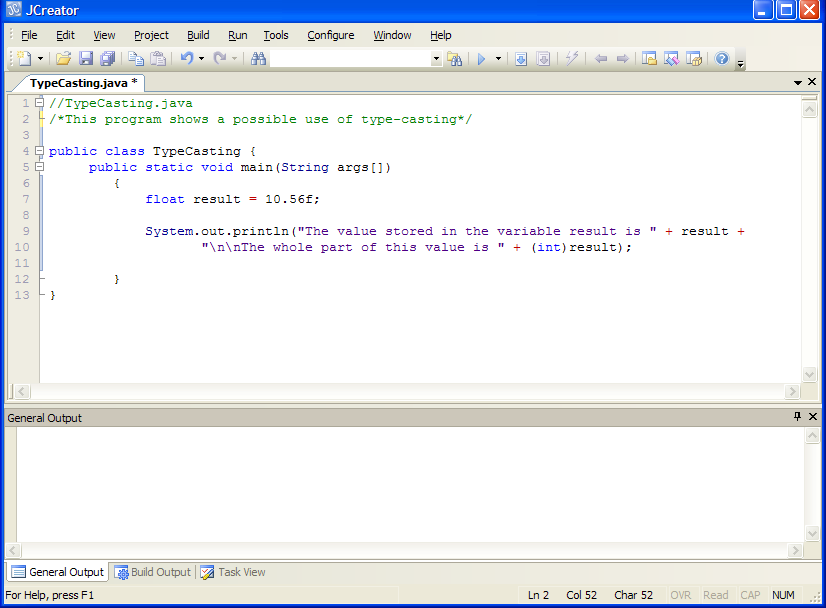
Now the first distance variable is type-cast to a float momentarily and so the problem is solved as the limit for float is extremely large. Running the program now yields the correct result:



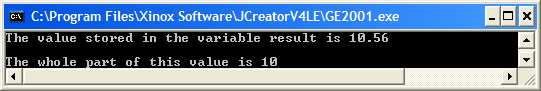
**Modify** your existing program now to take account of the type-casting. Note that it would have been just as acceptable to typecast the other distance variable instead here.

**More Uses Of Type-Casting**

Using type-casting got us out of trouble in the last program, but it has many other handy uses. For example, imagine that we have a program where we have a value stored in a floating-point variable called result. Now, if for some reason we just want to know the part of the value before the decimal point, we can use type-casting to quickly find it as follows:



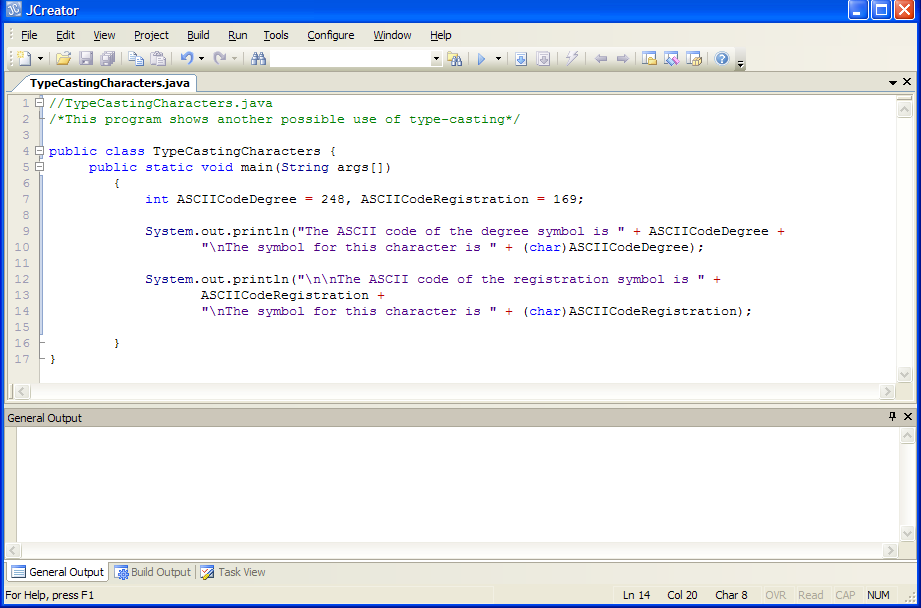
Here we typecast the variable result as an integer to get at the whole part of the number. This program runs as follows:



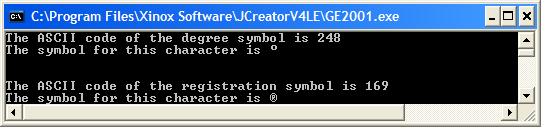
You **don’t need to type in the above program** unless you wish to for practice.

You will get an opportunity to use this to your advantage in some exercises later on in the module.

Yet another way to take advantage of type-casting is to use it to display the values of certain characters that are not available to us directly on the keyboard. We can access all of the alphabet, the digits and many symbols such as $, %, & etc. directly from the keyboard but others such as the symbol for degrees i.e. º and the registration symbol ® are not directly available. There is a way around this through type-casting however. Every one of the characters associated with the keyboard form part of the **ASCII character set** and each has a unique number e.g. ‘A’ has an ASCII code of 65, ‘3’ has an ASCII code of 51 etc. As it happens, the ASCII code for the degree symbol is 248 and that for the registration symbol is 169, which are part of the ***extended* ASCII character set** (anything above 127 is part of this set). The following code shows how type-casting can be used in this situation to display these characters (again, you don’t need to type in this demo program unless you wish to)

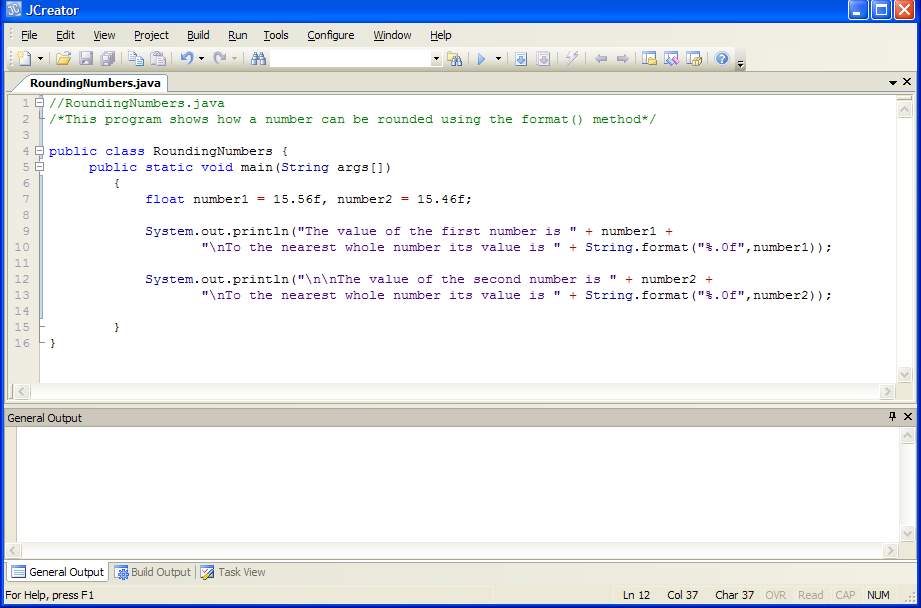


Here we typecast the integer variables ASCIICodeDegree and ASCIICodeRegistration as **char** to basically convert them on the fly to their character equivalents. The program runs as follows in the console window (note that if you run this program yourself you might experience different results if you are viewing the output in the “General Output” window – this is because the window uses a different font to the “terminal” font used by the console window. You can change the font used by the “General Output” window to “terminal” by selecting **Configure->Options->Workspace->Output View->Font** and selecting “Terminal” from the list). We will discuss the **char** data type and the ASCII character set in a much more detail in next weeks lab sheet. For now it is handy to know something useful about them.

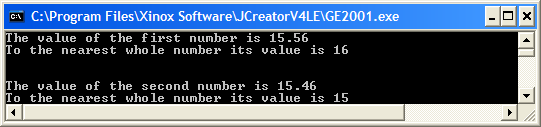


**Rounding Numbers in Java**

Rounding a floating-point number to the nearest whole number is easy. We can use the String.format() method and use **"%.0f"** as the format specifier. It just means correct to 0 decimal places i.e. round to the nearest whole number so 4.49 rounds down to 4 and 4.5 rounds up to 5. The following code illustrates how this can be done:



And it runs as follows:

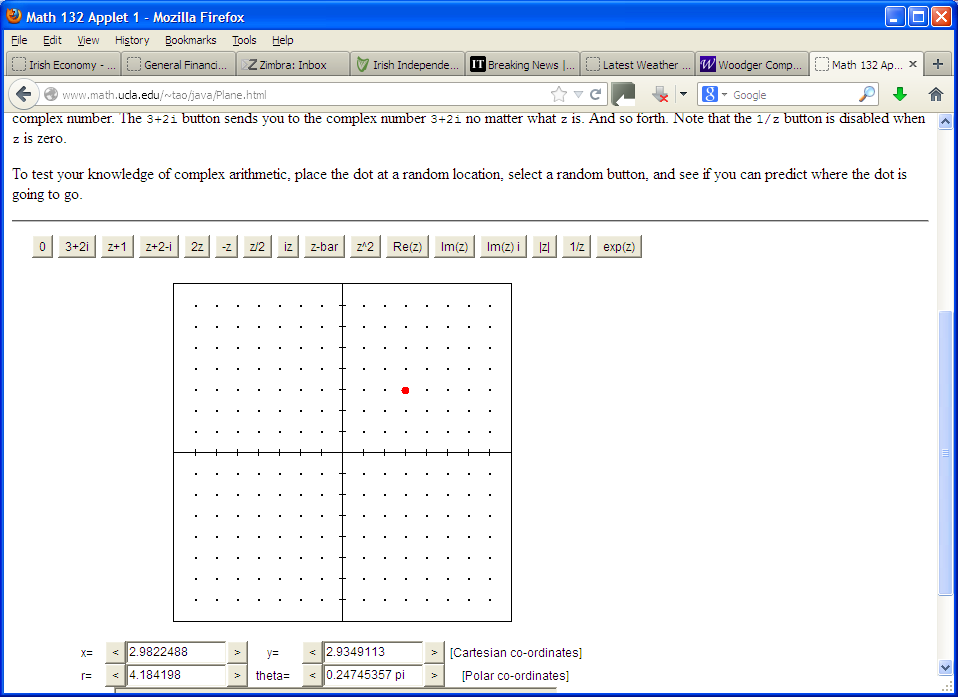


**Java in the Real World**

If you have gotten through the lab sheet and have some time to spare, it’s worth checking out what Java is capable of in the real world. Go to the following link now to see how Java is being used by Maths professor Terence Tao in the US to teach some mathematical concepts that are part of his course. The programs he uses are called Java “**applets**” and these are a special type of Java program that are **designed to run inside a web browser**. His applets web page is at

<http://www.math.ucla.edu/~tao/java/index2.html>

You will see a number of links to various applets, which can be clicked on and loaded into your web browser. As you will most likely have covered complex numbers at some stage for your leaving cert (don’t worry if you haven’t), click on “Applet 1” – “**The complex plane**” to see the applet in operation. You can interact with it and see how good your knowledge of complex numbers is by messing around with the graph of the complex plane.



The **source Java code** for these applets is also available for download. I have downloaded the code for this complex numbers applet for you to have a quick look at. In particular, as this lab sheet has dealt with arithmetic and calculations in Java, you should take a look at the code in **Complex.java**, which I have placed in the **ComplexNumbersApplet** folder within the **Lab3** folder for you. Don’t worry about how the code is organized here, or the fact that there is no main() in the file. You will understand the reasons for this once you have completed semester 2 OOP1. Just have a look at the code overall to see what you can recognise, see how there is a **public** **class** called **Complex** created. See that there are **float** variables called x and y declared. See how the method plus()

**public Complex plus(Complex z)**

**{**

**if (infinite || z.infinite) return new Complex(true);**

**return new Complex(x + z.x, y + z.y);**

**}**

is used to **add** 2 complex numbers – it does this with the expression **(x+z.x, y+z.y)** using the **addition operator**.

The method

**public float abs()**

**{**

**return (float) Math.sqrt(x\*x + y\*y);**

**}**

Is used to find the **absolute value** of a complex number. It does this using the **Math.sqrt()** method which gives the **square root** of a number. See how the **addition** and **multiplication** operators and **type-casting** is being used here also.

There are lots of other methods defined that do other jobs, many of which use **predefined methods** from the **Math** class. You will meet some of these methods later in the module. You will spend a lot of time in semester 2 creating your own **user-defined** methods. For now, you will simply continue using **predefined methods** such as print(), format(), nextInt() etc.

It is great practice to look at a program like this one, just to see what you can recognize and to see if you can understand any of the code that it contains. It also shows the power of the programming language and its **flexibility**.