Question 1.1

Welcome to Learning C# with Unity

First, let us start with what exactly C# is, C# is a high-level language that is statically and strongly typed. C# was created by Microsoft in 2000 and is part of the C family of coding.

Now we understand the type of language that C# is we can continue to build on this knowledge. First, we are going to output a string to the console, however, if you run the code below, you will encounter errors, this is due to no semicolon being at the end of the statement meaning the compiler does not know this is the end of the statement. Building upon this, the correct spelling of functions and variables is paramount else the compiler does not know what you are referencing. There are two of these errors in the debug.log function, if you succeed in correcting them you should see ‘Hello World’ outputted on the right of your screen.

1. debug.log(“Hello World”)

**A statically typed language is one in which the type of a variable is known at compile-time instead of at run-time. An example of this would be having an integer value called 'num' this value always needs to be an integer and cannot change type to some other type like a float or a string.**

**A strongly-typed programming language is one in which each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types.**

**In C# there are 5 base variables which are then built upon to create more variables, these five variables are:**

1. **int – stores integer (whole numbers) values such as 123 or -123 and has a maximum value of 2147483647 and a minimum value of -2147483648**.
2. **double – stores floating point values such as 19.99 or -19.99 and doubles are accurate up to 16 decimal places.**
3. **char – stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes.**
4. **string –** **stores text, such as "Hello World". String values are surrounded by double quotes.**
5. **bool – stores values with two states: true or false**

**As mentioned earlier these are not the only variables within C#, as you can see int has a min and max value, if we exceed these values will we get errors in our program, to counter this a ‘long’ can be used. A long stores integer values just like the int but has a maximum and minimum value of 9,223,372,036,854,775,807 and -9,223,372,036,854,775,808 respectively.**

Question 1.2

Now we can output to the console, we can start building our knowledge. First, we are going to add to numbers together, this is done like real life, but the main difference is a variable is defined with an integer value before numbers can be added together. In C#, it would look like this.

1. int a = 5, b = 5, c;
2. c = a + b;

The code above would give us a result of 10, Try out the above code and see if you can find other ways of adding numbers together.

Question 1.3

Subtraction of integers works nearly the same as the addition of integers. In this lesson you will see a new technique of subtracting numbers which may also be used to add numbers.

Instead of subtracting two integers from one another, we could subtract by doing the following:

1. int a = 5;

2. a -= 5;

This code is saying that the new value of 'a' is equal to the current value of 'a' minus 5.

Try out the code and output the result to the console.

Question 1.4

Multiplication of integers works just like the real world; in C# we use the asterisk (\*) to represent a multiplication. Additionally, multiplication within C# has to follow the same PEMDAS system as the real world. For example, the following code would output a result of 30, this is because the multiplication is being done first and the compiler is calculating 5 + 25.

1. int a = 5 + 5 \* 5;

However, if we add a pair of parentheses around 5 + 5 to make the code look like this:

1. int a = (5 + 5) \* 5;

We would now get a result of 50 due to the compiler performing the operation in the parenthesis first.

Now we understand multiplication, try placing a pair of parentheses in this code so it no longer outputs 34 but instead outputs 64.

**PEMDAS is an acronym for Parentheses, Exponential, Multiplication, Division, Addition and Subtraction. In the United Kingdom this system may be know as BODMAS (Brackets, Orders, Division, Multiplication, Addition, Subtraction). Both systems can be used as the work the same and is to be used when there are two or more operations in a single expression.**

Question 1.5

The division of integers follows the same rules as the multiplication, in C# we represent a division with the forward slash (/). Division still follows the order of operation formally mentioned in the previous question. As there is not much more to say on division why not try solving the following equation so it so it no longer outputs -4 but now -1

1. int a = -2 \* 1 \* 4 - 2 / 2 + 6 + 2 - 3;

Question 1.6

The Modulus function is used to calculate the remainder between two integer values, this is donated in C# with the percentage sign, in code may look like this:

1. int a = (5 % 2);

This code above is checking how many times 2 can go into 5, which is two times with a remainder of one. If you were wondering how this is done, the compiler reads the above line using the following equation: a – ( a / b ) \* b. If we were to plug in our current numbers the equation would look like the following: 5 – ( 5 / 4 ) \* 4 and we would have a remainder of 1.

Although 5 / 4 will equal 1.25, as these are integers the value will always be rounded to the nearest whole value that is less than the original value, so in our case this is 1.

Run the following code and you will see that using the equation of the modulus will yield the same result.

Question 1.7

The addition and subtraction of floats works both the same as in real life and how we add integers together in the previous section, the only difference is a floating-point number needs an ‘f’ at the end of the number so the compiler knows that it is a floating-point number. However, there is a problem with using floats and that is due to then fact they are only accurate to, on average, eight decimal places. This is due to a float being stored using only 4 bytes and anything larger than this will give floating point errors.

If you run the following code the second answer will be incorrect, the answer should 0.2463574474 but due to the answer being stored as a float the last three decimal places are rounded to the 7th decimal place (or the eight bit).

Question 1.8

The modulus of floats is similar to the integer method but the equation now looks like the following:

|x| - n \* |y|

Where |x| and |y| are the absolute values of x and y, this means they need to be real numbers resulting in negative numbers being treated as positive and n the largest possible integer that is less than or equal to the value of |x| / |y|.

If you were to run the following code, you would see that the both methods do not produce the same result as the integer example and this is due to the second equation not rounding the float to a integer.