Mr. Bellavia

LESSON 03 - BEDMAS, INTEGER VARIABLES, & COMMENTS

In this lesson we will dive more into the use of BEDMAS in our calculations. We will also start to use variables, specifically integer variables. Finally, we will look at techniques for commenting your code.

Sections:

I. BEDMAS:

At the hardware level, your computer is essentially a very powerful mathematical calculator, and all math follows the acronym **BEDMAS (Brackets, Exponents, Division, Multiplication, Addition, and Subtraction).** This represents the order of operations. For example:

```
Console.WriteLine(2 + 3 * 10);
```

Output will be:

```
32
```

The answer is **not** '50' because of **BEDMAS:**

BEDMAS requires that **(M)ultiplication** occurs **before (A)ddition!** Therefore, **'3'** will get multiplied by **'10'** first which will equal **'30'** which then gets added to **'2'** resulting in **'32'** as the final answer:

```
= 2 + 3 * 10      (Multiplication first)
= 2 + 30          (Addition second)
= 32
```

Recall the mathematical symbols from the last lesson:

Mathematical Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/

Along with the above symbols, you can use brackets **( )**. **'Brackets'** is the first word in the acronym **BEDMAS** which means brackets will always be evaluated first. Consider the following:

```
Console.WriteLine((2 + 3) * 10);
```

Output will be:

```
50
```

Because of the brackets, the statement above will equate **'(2 + 3)'** first which equals **'5'** which will then be multiplied by **'10'** resulting in **'50'** as the final answer:

```
= (2 + 3) * 10      (Brackets first)

= 5 * 10            (Multiplication second)

= 50
```

We will not spend too much time on mathematical operations in this course, but at the least, remember that **BEDMAS** is crucial! Imagine rockets being sent to the moon where the computer software that was used during launch ignored BEDMAS!

II. INTEGER VARIABLES:

Using **variables** is a fundamental part of computer programming. Variables allow you to access computer memory (i.e., **store information**). For example, consider the following code where we output a **'5'** to the **terminal:**

```
Console.WriteLine(5);
```

Or where we output calculations:

```
Console.WriteLine(3 * 3 + (1 - 4));
```

What if you wanted to use these numbers/calculations again later in your code? You can do so by **storing** them into **variables**. Variables can take on many values. **Integer Variables** store integer values. For example, whole numbers like **'3',** and not decimal values like 3.2. Let us rewrite the above code to demonstrate the use of an **integer variable:**

```
int x = 5;
Console.WriteLine(x);
```

Output will be:

```
5
```

And where we have:

```
int y = (2 + 3) * 10;
Console.WriteLine(y);
```

Output will be:

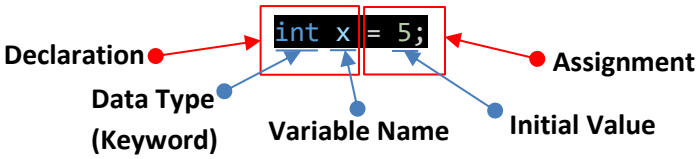
```
50
```

Both **'x'** and **'y'** are **integer variables.** We **assigned** an **integer value** to the variable named **'x'** and an **integer calculation** to the variable named '**y'** by using the equal sign **'='**, then we outputted each variable to the console using **'Console.WriteLine().** Notice the keyword **'int'** that was used before the variables **'x'** & **'y'. Keywords** are special words used in computer programming languages. In this case, the C# keyword **'int'** is used to create an **integer variable.**

Note: Many programming languages (not just C#) use **'int'** for creating integer variables. In fact, many programming languages share similar syntax altogether.

III. PROPER USE OF VARIABLES:

It is important to understand the process of creating and using a variable. Here is a breakdown:



We always start by **declaring** a variable with a specific **data type** keyword. In this case we are using the **integer data type** (we will see more types in the following lessons). Then you assign a **variable name,** in this case **'x'.** Finally, you **assign** an **initial value** to your variable in this case **'5'.**

If you do not assign a variable an **initial value** you will get a **build error**, for example:

```
int x;
Console.WriteLine(x);
```

Output will be a **build error!** Variable **'x'** has **no initial value!**

This is syntactically wrong since you did not assign **'x'** an **initial value.** You will get a **build error** if you try to run this code. Similarly, if you do not **declare** the data type of a variable you will also get a **build error:**

```
x = 5;
Console.WriteLine(x);
```

Output will be a **build error!** Missing the keyword **'int'.**

You can however **declare** a variable without giving it an initial value and **assign an initial value later in your code**, for example:

```
int x;
Console.WriteLine("Hello World!");
x = 5;
Console.WriteLine(x);
```

Output will be:

```
Hello World!
5
```

You can also **reuse** variables. However, you can only use the **keyword** (i.e., **'int'**) **once** for **each variable** (i.e., you cannot **redeclare** a variable)**.** For example, in the following code the **keyword 'int'** is used only **one** time for the **integer variable 'x'**, the **initial value** is not assigned until **line 2,** and we reuse the variable **'x' multiple times** after **(lines 3 & 4)**:

```
1  int x;
2  x = 5;
3  Console.WriteLine(x);
4  x = 5 + 1;
5  Console.WriteLine(x);
6  x = x + 1;
7  Console.WriteLine(x);
```

Output will be:

```
5
6
7
```

Also know that you **cannot declare** a variable with the **same name**. If you do, you will get a **build error:**

```
int x;
x = 5;
Console.WriteLine(x);
int x;
x = 5 + 1;
Console.WriteLine(x);
x = x + 1;
Console.WriteLine(x);
```

The second **'int x;'** will give you a **build error** when you try to run your program! **DO NOT DO THIS!**

Just like we did in the last lesson, you can mix and match strings and variables in the **same Console.WriteLine()** by separating them with a **'+'** sign. For example:

```
int x;
x = 5 + 5;
Console.WriteLine("5 + 5 = " + x);
```

String Literal **Integer Variable**

In our **Console.WriteLine()** statement above, we have combined two different **data types** (**strings** and **integers**).

Note: If a value is not **stored** inside a **variable** it is referred to as a **literal value** (for example**, string literals**, **integer literals**, etc.).

Also, notice how variables retain their value until you assign them a new value. For example:

```
1  x = 5;
2  Console.WriteLine(x);
3  x = 5 + 1;
4  Console.WriteLine(x);
5  x = x + 1;
6  Console.WriteLine(x);
```

Output will be:

```
5
6
7
```

Notice **after line 3** executes **'x'** gets **overwritten** and will **equal '6',** and then on **line 5** we have 'x = x + 1' which will equate to 'x = **6** + 1' (the variable **'x'** equals **itself (6) plus '1'**). When you add a variable to itself plus **'1'**, that variable is referred to as a **'counter'** variable**.** We will see the use of **counter variables** later in this course.

One last thing to note, multiple variables of the **same data type** (i.e, int) can be **declared** and **assigned** on **the same line.** For example:

```
int x = 0, y = 4;
```

Or you can **declare** the variables on the same line, then **assign** their **initial values** after, for example:

```
int x, y;
x = 0;
y = 4;
```

The possibilities are endless with how you construct your expressions. For example:

```
1  int x;
2  int y = 2;
3  x = 3;
4  Console.WriteLine(x + y);
5
6  int z = x + y;
7  Console.WriteLine(z);
8
9  z = 3 * 2 + y;
10 Console.WriteLine(z);
```

Output will be:

```
5
5
8
```

As you can see, we can construct **calculations** using **variables** inside our **mathematical expressions**. This gives us a lot of flexibility to construct very complex equations.

IV. VARIABLE NAMING:

You can choose any name you wish when **naming a variable.** However, there are a few rules:

- **Characters allowed:**
  - Underscore (_)
  - Uppercase letters (A – Z)
  - Lowercase letters (a – z)
  - Digits (0 – 9)

Note: There are also a small number of other special characters that can be used. However, certain special characters could change the nature of how your variables work in some programming languages, therefore it is wise to avoid using special characters at this point.

To demonstrate, the following examples are **allowed:**

```
int x = 1;
int apples = 3;
int myFirstInteger;
int y2 = 0;
int my_int;
```

Also notice how we named the third variable above **('myFirstInteger').** The first word 'my' is all lowercase and each word after ('First' and 'Integer) are all lower case except for the respective first letter. This style of variable naming is called **camel casing** and you should utilize it as much as possible when your variable contains multiple words. This will keep your code clean and more readable. If your variable only consists of one word, for example **'apples',** then you can keep it all lowercase.

There are also rules for characters and spellings that are **not allowed:**

- **Spaces** are **not** allowed.
- **First letter cannot** start with a **number.**
- **Keywords** (like 'int') **cannot** be used as a variable name.

To demonstrate, the following examples are **not allowed:**

```
int int = 1;
int 1x;
int my integer = -1;
int test#;
```

V. COMMENTS:

Keeping your code organized will be very important once your program contains multiple lines of code. We can create notes in our code using **comments.** Comments are initiated with two forward slashes **'//'.** For example:

```
1  // a simple calculation
2  int x = 5 + 3;
3  Console.WriteLine(x);
```

Output will be:

```
8
```

In the above code the comment on **line 1** is a **comment** has **no effect** on your running code. You can use as many comments in your code as you wish.

You can also use **block comments** to comment out a bunch of code at once. For example:

```
1  int length;
2  int width;
3  /*length = 5;
4  width = 3;*/
5  length = 2;
6  width = 6;
7  int area = length * width;
8  Console.WriteLine(area);
```

Output will be:

```
12
```

In the above code on **line 3** we use a forward slash followed by an asterisk **'/*'** to **start** our block comment, and on **line 4** we use an asterisk followed by a forward slash **'/*'** to **end** our block comment. Everything in between **'/*'** and **'*/'** will be commented out and ignored at runtime. Also notice how Visual Studio will show comments in green.

Block comments can be very useful when you are trying to **debug** your code (i.e., finding errors in your code). They are also useful when you do not want to delete any lines of code that you may be replacing for testing purposes.