CHEAT SHEET ON COMMON ERRORS AND THEIR MEANING THAT YOU CAN PRINT AND KEEP CLOSE BY ☺

COMMON ERRORS AND THEIR MEANING

As you will start your journey through C# coding, you may sometimes find it difficult to interpret the errors produced by Unity in the console. However, after some practice, you will manage to recognize them, to understand (and also avoid) them, and to fix them accordingly. The next list identifies the errors that my students often come across when they start coding in C#.
When an error occurs, Unity usually provides you with enough information to check where it has occurred, so that you can fix it. While many are relatively obvious to spot, some others are trickier to find. In the following, I have listed some of the most common errors that you will come across as you start with C#. The trick is to recognize the error message so that you can understand what Unity is trying to tell you. Again, this is part of the learning process, and you **WILL** make these mistakes, but as you see these errors, you will learn to understand them (and avoid them too :-)). Again, Unity is trying to help you by communicating, to the best that it can, where the issue is; by understanding the error messages we can get to fix these bugs easily. So that it is easier to fix errors, Unity usually provides the following information when an error occurs:

- Name of the script where the error was found.
- The number of the row and column where the error was found.
- A description of the error that was found.

So, if Unity was to generate the following message **"Assets/Scripts/MyFirstScript.cs (23,34) BCE0085: Unknown identifier: 'localVariable'"**, it is telling us that an error has occurred in the script called **MyFirstScript**, at the line **23**, and around the **34th** character (i.e., column) on this line. In this particular message, it is telling us that it can't recognize the variable **localVariable**.
So, you may come across the following errors (this list is also available in the resource pack as a pdf file, so that you can print it and keep it close by):

- **";" expected**: This error could mean that you have forgotten to add a semi-colon at the end of a statement. To fix this error, just go to the line mentioned in the error message and ensure that you add a semi-colon.
- **Unknown identifier**: This error could mean that Unity does not know the variable that you are mentioning. It can be due to at least three reasons: (1) the variable has not been declared yet, (2) the variable has been declared but outside the scope of the method (e.g., declared locally in a different function), or (3) the name of the variable that you are using is incorrect (i.e., spelling or case). Remember, the names of all variables and functions are case-sensitive; so by just using an incorrect case, Unity will assume that you refer to another variable, which, in this case, has not been declared yet.
- **The best method overload for function … is not compatible**: This error is probably due to the fact that you are trying to call a function and to pass a parameter with a type that is not what Unity is expecting. For example, the method **mySecondMethod**,

described in the next code snippet, is expecting a **String** value for its parameter; so, if you pass an integer value instead, an error will be generated.

```
void mySecondFunction(string name)
{
	print (“Hello, your name is” +name);
}
mySecondFunction(“John”);//this is correct
mySecondFunction(10);//this will trigger an error
```

- **Expecting } found …:** This error is due to the fact that you may have forgotten to either close or open curly brackets. This can be the case for conditional statements or functions. To avoid this issue, there is a trick (or best practice) that you can use: you can ensure that you indent your code so that corresponding opening and closing brackets are at the same level. In the next example, you can see that the brackets corresponding to the start and end of the method **testBrackets** are indented at the same level, and so are the brackets for each of the conditional statements within this function. By indenting your code (using several spaces or tabulation), you can make sure that your code is clear and that missing curly brackets are easier to spot.

```
Void testBrackets()
{
	if (myVar == 2)
	{
		print ("Hello World”);
		myVar = 4;
	}
	else
	{
	}
}
```

Sometimes, although the syntax of your code is correct and does not yield any error in the **Console** window, it looks like nothing is happening; in other words, it looks like the code, and especially the methods that you have created do not work. This is bound to happen as you create your first scripts. It can be quite frustrating (and I have been there :-)) because, in this case, Unity will not let us know where the error is. However, there is a succession of checks that you can perform to ensure that this does not happen; so you could check the following:

- The script that you have written has been saved.
- The script has no errors.
- The script is attached to an object.
- If the script is indeed attached to an object and you are using a built-in method that depends on the type of object it is attached to, make sure that the script is linked to the correct object. For example, if your script is using the built-in method **OnControllerColliderHit**, which is used to detect collision between the **FPSController** and other objects, but you don’t drag and drop the script on the **FPSController** object, the script, while being error-free, will not be used, and the method **OnControllerColliderHit** will not be called if you collide with an object.
- If the script is indeed attached to the right object and is using a built-in method such as **Start**, or **Update**, make sure that these functions are spelt properly (i.e., exact

spelling and case). For example for the method **Update**, what happens here is that the system will call the method **Update** every frame, and no other function. So if you write a method spelt **update**, the system will look for the **Update** function, and since it has not been defined (or overwritten), nothing will happen, unless you specifically call this function. The same would happen for the method **Start**. In both cases, the system will assume that you have created two new functions **update** and **start**.