**Understanding Basic Console Input and Output**

**The goal for this exercise** is to understand how to both print messages and data to the screen, as well as how to read information from the keyboard.

Download the starter project for this set of Post-Class Exercises, if you haven’t already. Look for the line that reads “class Basic\_Console\_IO”, and within that class, look for the method (aka function, aka subroutine, aka procedure, etc, etc) named RunExercise (it’ll start on a line that looks like this: “public void RunExercise()”). We will refer to this method as Basic\_Console\_IO **.**RunExercise –this will clearly identify it, and keep it separate from the RunExercise method of other exercises.

 You’ll notice that right now, it just has:

**Console.WriteLine("Hello, World!");**

**Console** represents the output window of a C# program. Using it in conjunction with the **WriteLine** method (a.k.a. function, command, etc) causes information from the program to be displayed to the screen. Because we are using the **WriteLine** command, it causes the cursor to move down to the beginning of the next line. If we had used the **Write** method instead, the cursor would have remained on the same line, so that the next output had appeared after this one. For example, the following code will accomplish the same thing as the above code:

**Console.Write("Hello, ");**

**Console.WriteLine("World!");**

Change the output statement in the file to the above in order to see for yourself that this is correct.

In addition to displaying information to the screen, there is a mechanism for reading information into a C# program entered at the keyboard by the program user. In order to use it, you must first become familiar with another important concept – declaring variables.

A **variable** is a memory location that has been reserved by your program to store information. Inside your program, it is referred to by its **variable name**. A C# statement called a **declaration** is used to declare a variable and give it a name.

In C#, every variable must have a specific **type**. A variable’s type dictates what it represents. Two fundamental types that you’ll be using a lot are **int**, used to represent integer values, and **double**, used to represent real numbers (numbers that can have a fractional component). Here is some sample code that declares one of each, gives them a value, and displays them:

|  |
| --- |
|  class Basic\_Console\_IO { public void RunExercise() { int x; double y; x = 4; y = 7.5; Console.WriteLine(x); Console.WriteLine(y); } } |

Replace the contents of Basic\_Console\_IO **.**RunExercise with the code above, rebuild and then rerun the program. What are the results?

The first two lines of the method

 **int x;**

 **double** **y;**

are the variable declarations. A declaration consists of a type followed by a variable name and then a semicolon. Semicolons follow **every** statement in C#. The compiler will not build a program that has omitted a semicolon at the end of a statement. Take out a semicolon from the program above and rebuild it. What happens?

The second two lines of the program

**x = 4;**

**y = 7.5;**

are called assignment statements. They put specific values into the variable locations. An assignment statement consists of a variable name, an equals sign, and then a value or an expression (you’ll learn more about expressions later) followed by a semicolon.

The last line of the program displays the values stored in the variables. Notice that you can put a variable name inside of a Console.WriteLine statement and the *value stored* in the variable will be displayed (as opposed to displaying the name of the variable – x or y in this case).

Note that we can also have a single WriteLine (or Write) statement print out multiple pieces of information on one line. If you want to do this we must separate each piece of information with the plus-sign (**+**), as per the following example:

 int x;

 double y;

 x = 4;

 y = 7.5;

 Console.WriteLine("X is: " + x + " and that's good");

 Console.WriteLine("Y is: " + y );

Note that we could have also written the above code as :

 int x;

 double y;

 x = 4;

 y = 7.5;

 Console.WriteLine("X is: **{0}** and that's good"**, x**);

 Console.WriteLine("Y is: **{0}**"**, y** );

Using this format, C# will replace the {0} in the message with the value of the first variable that’s listed after the message. In the first Console.WriteLine example, {0} will be replaced with the value currently stored in the variable **x**, while in the second example, {0} corresponds with y. For reasons that we’ll get into later in the term, this form (using {0} ) is preferable to the previous form (using the + operator to concatenate stuff), but you should be comfortable with either one.

The last concept covered in this section is input from the keyboard. In order to do this, we'll need to introducing another fundamental data typed: **string**, used to represent text information (this is short for 'a string/sequence of individual characters').

Update the contents of Basic\_Console\_IO **.**RunExercise with the following code:

public void RunExercise()

{

 string szInput;

 int x;

 double y;

 Console.Write("Type a value for X here: ");

 szInput = Console.ReadLine();

 Int32.TryParse(szInput, out x);

 Console.WriteLine("You typed: " + x);

 Console.Write("Type a value for Y here: ");

 szInput = Console.ReadLine();

 Double.TryParse(szInput, out y);

 Console.WriteLine("You typed: " + y);

}

|  |
| --- |
|  Notice that the above code uses the **TryParse** methods, whereas your book tends to use the **Convert.ToInt32** (or **Convert.ToDouble**) methods. The main advantage of TryParse is that it will return **false** if the user types in a bad number, while the Convert.\* methods will crash (by throwing an Exception). Using the TryParse methods will simplify your code, but you're welcome to use either method – personally, I'd use whichever one you feel more comfortable with. Also, you may notice that not all of my examples/samples use the TryParse method – if you notice this, let me know & I'll fix that up. Unless the example/sample states otherwise, my not using TryParse is coincidence ☺ |

Rebuild and run it. You’ll notice that after *“Type a value for X here: “* is displayed, nothing happens. The statement **Console.ReadLine** causes the program to wait for the user to enter an integer value at the keyboard and then press <ENTER>. Do this. Now the program finishes executing and displays the number that you just entered.

**Console.ReadLine** reads information from the keyboard, and stores the raw text into the variable named **szInput**. Note that at this point we only have the text that the user typed, but don't know what number it is (or if the user even typed a number). The line Int32.TryParse (or Double.TryParse) will attempt to convert that text into an integer (or a double, respectively). It expects the user to correctly enter the type of information indicated by the type of the variable that is its argument. If you don't type in the correct type of data, the program will crash, which for now is ok (when asked to type in an integer, try typing your name instead, and see what happens)

 **What you need to do for this exercise:**

1. You should have read everything above, understood everything above, and tried out all the code, on your own, so that you’re clear about how it works. You are also encouraged (as you always are) to ‘play around’ with this, as well – try stuff that isn’t specifically called for above, just to test your knowledge, and just to see if you can.
2. In particular, you need to know the difference between **Write** and **WriteLine**.
3. Other than that, there are no particular requirements for this exercise.