PREPARING FOR PROGRAMMING ASSIGNMENT #1

Prompting the User for Input

Be sure to provide clear, helpful instructions to your users about what they need to do to use your program. Use prompts such as "Please enter a number between 0 and 100:"

Also, GRAMMAR COUNTS!   Take the extra 20 seconds needed to make your user prompts grammatically correct.

Reading User Input

Use Console.ReadLine() to get input from your users. Note that initially, that input will be given to you as a string.

Converting Strings into Numbers

There are several ways to do this. At this point we only want to use the "int.TryParse()" method. Note that this is not the perferred way to parse a string into an integer (and so, if you look on StackExchange you will find some different approaches) however we haven't learned everything we need to know in order to use those other methods. Thus we cannot (and you should not) use those other methods in your homework! int.TryParse() doesn't require that extra stuff - and so it will do just fine for our purposes at this point.

Here is the C# documentation for int.TryParse():   **https://docs.microsoft.com/en-us/dotnet/api/system.int32.tryparse?view=netframework-4.8 (https://docs.microsoft.com/en-us/dotnet/api/system.int32.tryparse?view=netframework-4.8)**

TryParse(String, Int32)

Converts the string representation of a number to its 32-bit signed integer equivalent. A return value indicates whether the conversion succeeded.

```
    public static bool TryParse (string s, out int resul
t);
```

Parameters

String (https://docs.microsoft.com/en-us/dotnet/api/system.string?view=netframework-4.7.2) s -

A string containing a number to convert.

out Int32 (https://docs.microsoft.com/en-us/dotnet/api/system.int32?view=netframework-4.7.2) result -

A variable for storing the result of the conversion. If the conversion succeeded, it will contain the 32-bit signed integer value equivalent of the number contained in `s`. If the conversion failed it will contain zero. The conversion fails if the `s` parameter is `null` or Empty (https://docs.microsoft.com/en-us/dotnet/api/system.string.empty?view=netframework-4.7.2) , is not of the correct format, or represents a number less than MinValue (https://docs.microsoft.com/en-us/dotnet/api/system.int32.minvalue?view=netframework-4.7.2) or greater than MaxValue (https://docs.microsoft.com/en-us/dotnet/api/system.int32.maxvalue?view=netframework-4.7.2) . This parameter is passed uninitialized; any value originally supplied in `result` will be overwritten.

Returns:

Boolean - (https://docs.microsoft.com/en-us/dotnet/api/system.boolean?view=netframework-4.7.2)

`true` if `s` was converted successfully; otherwise, `false`.

Note that TryParse() takes two parameters - the string to convert and the integer to put the converted number into. Also, note that you have to add the keyword "out" in front of the second parameter. For example, this code snippet would convert the string "100" into an integer and store it in the variable "n" - **int.TryParse("100", out n);**

Here's an example of how to use int.TryParse() in a program:

```
using System;

namespace LWTech.ChipAnderson.HelloWorldApp
{
    class Program
    {
        static void Main(string[] args)
        {
            int age = 0;
            bool itWorked = false;
            string response = "";

            Console.WriteLine("Please enter your age:");

            while (!itWorked)
            {
                response = Console.ReadLine();

                itWorked = int.TryParse(response, out age);

                if (itWorked == false)
                {
                    Console.WriteLine("I'm sorry. I couldn't understand what you entered.  Please enter you age again usin
g just the number keys on your keyboard.");
                }
            }
            Console.WriteLine($"Your age is {age}");

            Console.ReadLine();
        }
    }
}
```

Dealing with Invalid Input

As you may have noticed in the program above, whenever we deal with users, we have to handle the case where they enter incorrect data. That means that whenever we are prompting for input, we need to do so in a loop so that we can re-prompt for input if they enter incorrect data. Again, you can see an example of how that works in the code snippet above. Notice however that the snippet doesn't fully validate the user's input.

While it will correctly re-prompt for the user's age if the user enters a non-integer, it doesn't check to see if the age entered is realistic.

Try it yourself!

Improve the program above so that it re-prompts the user if the user enters an invalid integer for their age.
After writing and testing your solution, check out this approach: **[PA01a] Try it yourself: Answer (https://lwtech.instructure.com/courses/1841516/pages/pa01a-try-it-yourself-answer)**

Separating the User-Interface from the Calculation Logic

As you create this program, you'll start to create "calculation" methods to help you. Try not to put user-interface code (i.e. Console methods) into any of those calculation methods. Instead, keep all the calls to Console.ReadLine() and Console.WriteLine() in your Main() method. While doing this might some bothersome at first, later we will see that keeping the user-interface code seperate from the calculation logic will make our programs much more readable, flexable and maintainable.

Modulo Arithmetic

When you need to see what the remainder is after you divide two integers, use the "%" operator. i.e. 98 % 60 = 38

Exponents

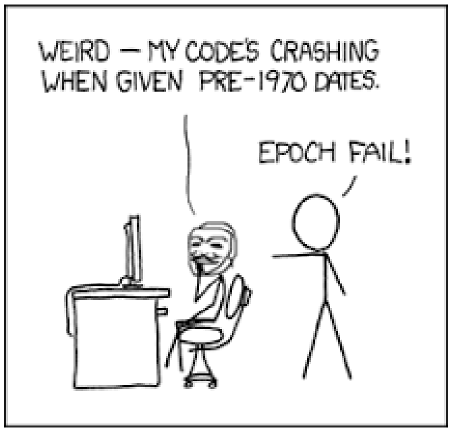
When you need to raise a number to a higher power - like when you want to calculate the square of some number - use the Math.Pow() method.

ints vs longs

Don't forget to think about the range of values that each of your variables might have to store. If you cannot guarantee that a number will be less than the largest value a 32-bit integer can store (i.e., 2,147,483,647) then you'll need to use a long (i.e. a 64-bit integer) instead.

Bulletproof!

When writing real-world programs, your code has to handle any data that any user can throw at it - even malicious and deliberately malformed data. Our programs have to be bulletproof! They have to take any conceivable kind of data and not crash and not generate incorrect or misleading output. Getting into the habit of making your programs bulletproof is a good thing to start doing right now. What other ways could your homework program be "attacked" by invalid input? How can you prevent bad data from crashing your program?



Debugging

If/when things go wrong, don't immediately jump into the Visual Studio Debugger.   As strange as it might seem, the debugger should not be used for your initial debugging work. Instead, try two other approaches:

1.) Execute the code in your head, using a pad of paper for "storing" variable values.

2.) Add Console.writeln() statements to your program to validate that things are working like you expect.

We'll talk more about debugging next week in class, but for now, try to stay away from the VS Debugger - it's too complex and overpowered for our needs right now.