

# Validating Inputs

<https://csci-1301.github.io/about#authors>

July 5, 2021 (02:42:21 PM)

## Contents

1	Using TryParse	1
2	Validating Inputs	1
3	Input validation with classes	2

## 1 Using TryParse

In your IDE, write statements using `int.TryParse` and `double.TryParse` to determine if values in the table below can be successfully parsed.

For each input:

1. in returns column, write whether the TryParse operation succeeded (**true** or **false**).
2. in value column, write the obtained numeric value after TryParse operation.

First few lines are given as examples, your task is to complete the rest of the table.

Input	int.TryParse		double.TryParse	
	returns	value	returns	value
"9432.0"	<b>false</b>	0	<b>true</b>	9432.0
"160519"	<b>true</b>	160519	<b>true</b>	160519
"12,804"				
" +5102"				
"2+2"				
" -322 "				
" (72) ; "				
"000"				
"78 095"				

**Question:** After completing the table, can you detect a pattern between “returns” and “value”?

## 2 Validating Inputs

For the following problems, always perform these steps:

- ask the user for input
- then check that the input is valid according to the specific problem
- *only then* perform the subsequent action.

If the provided input is not valid, request new input from user until user provides valid input.

1. Write a loop that displays: **Enter yes to quit:** then check the user's input. Consider any of these variations to mean yes: "yes", "YES", "y", "Y". Once user enters yes, exit the loop.
2. Ask the user to enter a positive integer, between 2 and 100 (including these boundary values 2 and 100). Make sure the value user enters is between these bounds. Then compute the sum of integers starting from 1 up to the integer user entered, and finally display that sum. Here are examples:
  - if user enters 5, compute:  $1 + 2 + 3 + 4 + 5$ , then display 15 at the screen
  - if user enters 8, compute:  $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$ , then display 36 at the screen
3. Do this next problem using **decimal** type. Ask the user to enter any numbers (can be positive, negative, or zero). Ignore all non-numeric inputs. Choose an appropriate sentinel value to enable user to indicate when they are done. Compute and display the average of all numbers user entered. If user entered no numbers, display "You did not enter any numbers".

### 3 Input validation with classes

Start by downloading the `LoanCalculator`<sup>1</sup> solution, that mixes classes and decision structures. Spend some time studying the implementation to understand *what* the program is doing and *how* it is doing it.

Next edit the `Program.cs` file of the `LoanCalculator` to add the following validation features:

1. Users entering a value other than A, a, H, h, 0 and o for the loan type will be asked again, and asked as long as they do not give a valid answer.
2. Users entering a credit score not between 300 and 850, or that is not an integer, will be asked again, and asked as long as they do not give a valid answer.
3. Users entering an amount needed or a down payment that is not a decimal, or is a negative decimal, will be asked again, and asked as long as they do not give a valid answer.
4. (Optional) Use the `ToLower()`<sup>2</sup> or `ToUpper()`<sup>3</sup> methods of the `char` class to make the program more readable – you will be able to greatly simplify the `if` statement that checks the loan type.
5. (Optional, hard) Write a method for the `Loan` class that takes a character as an argument, and returns the string describing the type of loan designed by that character. Then, use this method in the `ToString` and in the application program instead of doing it "by hand".

You can find a possible solution in this archive<sup>4</sup>.

---

<sup>1</sup>`LoanCalculator.zip`

<sup>2</sup><https://docs.microsoft.com/en-us/dotnet/api/system.char.tolower?view=netframework-4.7.2>

<sup>3</sup><https://docs.microsoft.com/en-us/dotnet/api/system.char.toupper?view=netframework-4.7.2>

<sup>4</sup>`Solution_LoanCalculator.zip`