2024-09-19

# Input Validation

## Valid and invalid data

* Depending on the purpose of your program, each variable might have a limited range of values that are “valid” or “good,” even if the data type can hold more
* For example, a decimal variable that holds a price (in dollars) should have a positive value, even though it is legal to store negative numbers in a decimal
* Consider the Item class, which represents an item sold in a store. It has a price attribute that should only store positive values:

class Item  
{  
 private string description;  
 private decimal price;  
  
 public Item(string initDesc, decimal initPrice)  
 {  
 description = initDesc;  
 price = initPrice;  
 }  
  
 public decimal GetPrice()  
 {  
 return price;  
 }  
  
 public void SetPrice(decimal p)  
 {  
 price = p;  
 }  
  
 public string GetDescription()  
 {  
 return description;  
 }  
  
 public void SetDescription(string desc)  
 {  
 description = desc;  
 }  
}

* When you write a program that constructs an Item from literal values, you (the programmer) can make sure you only use positive prices. However, if you construct an Item based on input provided by the user, you cannot be certain that the user will follow directions and enter a valid price:

Console.WriteLine("Enter the item's description");  
string desc = Console.ReadLine();  
Console.WriteLine("Enter the item's price (must be positive)");  
decimal price = decimal.Parse(Console.ReadLine());  
Item myItem = new Item(desc, price);

In this code, if the user enters a negative number, the myItem object will have a negative price, even though that does not make sense.

* One way to guard against “bad” user input values is to use an if statement or a conditional operator, as we saw in the previous lecture (Switch and Conditional), to provide a default value if the user’s input is invalid. In our example with Item, we could add a conditional operator to check whether price is negative before providing it to the Item constructor:

decimal price = decimal.Parse(Console.ReadLine());  
Item myItem = new Item(desc, (price >= 0) ? price : 0);

In this code, the second argument to the Item constructor is the result of the conditional operator, which will be 0 if price is negative.

* You can also put the conditional operator inside the constructor, to ensure that an Item with an invalid price can never be created. If we wrote this constructor inside the Item class:

public Item(string initDesc, decimal initPrice)  
{  
 description = initDesc;  
 price = (initPrice >= 0) ? initPrice : 0;  
}

then the instantiation new Item(desc, price) would never be able to create an object with a negative price. If the user provides an invalid price, the constructor will ignore their value and initialize the price instance variable to 0 instead.

## Ensuring data is valid with a loop

* Another way to protect your program from “bad” user input is to check whether the data is valid as soon as the user enters it, and prompt him/her to re-enter the data if it is not valid
* A while loop is the perfect fit for this approach: you can write a loop condition that is true when the user’s input is *invalid*, and ask the user for input in the body of the loop. This means your program will repeatedly ask the user for input until he/she enters valid data.
* This code uses a while loop to ensure the user enters a non-negative price:

Console.WriteLine("Enter the item's price.");  
decimal price = decimal.Parse(Console.ReadLine());  
while(price < 0)  
{  
 Console.WriteLine("Invalid price. Please enter a non-negative price.");  
 price = decimal.Parse(Console.ReadLine());  
}  
Item myItem = new Item(desc, price);

* The condition for the while loop is price < 0, which is true when the user’s input is invalid
* If the user enters a valid price the first time, the loop will not execute at all – remember that a while loop will skip the code block if the condition is false
* Inside the loop’s body, we ask the user for input again, and assign the result of decimal.Parse to the same price variable we use in the loop condition. This is what ensures that the loop will end: the variable in the condition gets changed in the body.
* If the user still enters a negative price, the loop condition will be true, and the body will execute again (prompting them to try again)
* If the user enters a valid price, the loop condition will be false, so the program will proceed to the next line and instantiate the Item
* Note that the *only* way for the program to “escape” from the while loop is for the user to enter a valid price. This means that new Item(desc, price) is guaranteed to create an Item with a non-negative price, even if we did not write the constructor that checks whether initPrice >= 0. On the next line of code after the end of a while loop, you can be certain that the loop’s condition is false, otherwise execution would not have reached that point.

## Ensuring the user enters a number with TryParse

* Another way that user input might be invalid: When asked for a number, the user could enter something that is not a number
* The Parse methods we have been using assume that the string they are given (in the argument) is a valid number, and produce a run-time error if it is not
* For example, this program will crash if the user enters “hello” instead of a number:

Console.WriteLine("Guess a number"):  
int guess = int.Parse(Console.ReadLine());  
if(guess == favoriteNumber)  
{  
 Console.WriteLine("That's my favorite number!");  
}

* Each built-in data type has a **TryParse method** that will *attempt* to convert a string to a number, but will not crash (produce a run-time error) if the conversion fails. Instead, TryParse indicates failure by returning the Boolean value false
* The TryParse method is used like this:

string userInput = Console.ReadLine();  
int intVar;  
bool success = int.TryParse(userInput, out intVar);

* The first parameter is a string to be parsed (userInput)
* The second parameter is an **out parameter**, and it is the name of a variable that will be assigned the result of the conversion. The keyword out indicates that a method parameter is used for *output* rather than *input*, and so the variable you use for that argument will be changed by the method.
* The return type of TryParse is bool, not int, and the value returned indicates whether the input string was successfully parsed
* If the string userInput contains an integer, TryParse will assign that integer value to intVar and return true (which gets assigned to success)
* If the string userInput does not contain an integer, TryParse will assign 0 to intVar and return false (which gets assigned to success)
* Either way, the program will not crash, and intVar will be assigned a new value
* The other data types have TryParse methods that are used the same way. The code will follow this general format:

bool success = <numeric datatype>.TryParse(<string to convert>, out <numeric variable to store result>)

Note that the variable you use in the out parameter must be the same type as the one whose TryParse method is being called. If you write decimal.TryParse, the out parameter must be a decimal variable.

* A more complete example of using TryParse:

Console.WriteLine("Please enter an integer");  
string userInput = Console.ReadLine();  
int intVar;  
bool success = int.TryParse(userInput, out intVar);  
if(success)  
{  
 Console.WriteLine($"The value entered was an integer: {intVar}");  
}  
else  
{  
 Console.WriteLine($"\"{userInput}\" was not an integer");  
}  
Console.WriteLine(intVar);

* The TryParse method will attempt to convert the user’s input to an int and store the result in intVar
* If the user entered an integer, success will be true, and the program will display “The value entered was an integer:” followed by the user’s value
* If the user entered some other string, success will be false, and the program will display a message indicating that it was not an integer
* Either way, intVar will be assigned a value, so it is safe to write Console.WriteLine(intVar). This will display the user’s input if the user entered an integer, or “0” if the user did not enter an integer.
* Just like with Parse, you can use Console.ReadLine() itself as the first argument rather than a string variable. Also, you can declare the output variable within the out parameter, instead of on a previous line. So we can read user input, declare an int variable, and attempt to parse the user’s input all on one line:

bool success = int.TryParse(Console.ReadLine(), out int intVar);

* You can use the return value of TryParse in a while loop to keep prompting the user until they enter valid input:

Console.WriteLine("Please enter an integer");  
bool success = int.TryParse(Console.ReadLine(), out int number);  
while(!success)  
{  
 Console.WriteLine("That was not an integer, please try again.");  
 success = int.TryParse(Console.ReadLine(), out number);  
}

* The loop condition should be true when the user’s input is *invalid*, so we use the negation operator ! to write a condition that is true when success is false
* Each time the loop body executes, both success and number are assigned new values by TryParse