



MALMÖ HÖGSKOLA

Programming Using C#, Basic Course

Assignment 3

GUI and Methods with parameters

Cinema Booking System (CBS)

Version 1- GUI and Input

Help and Guidance

[Farid Naisan](#)

University Lecturer
Department of Computer Science

Malmö University, Sweden

(Consider this document as part of the lectures in this module!)



Table of Contents

1. Overview	3
2. Class Diagram	4
3. Input and output.....	7
4. Work plan.....	8
5. Use Windows Forms Controls to design your GUI.....	9
6. The InputUtility class	12
7. Back to MainForm.....	15
8. Input reading using out-parameters (higher grade requirement)	18
9. Saving user input and displaying output.....	18
10. UpdateGUI.....	20
11. String.Format	21
12. What to do when the RadioButtons are clicked?.....	23
13. To set focus to and highlight the contents of a TextBox.....	23

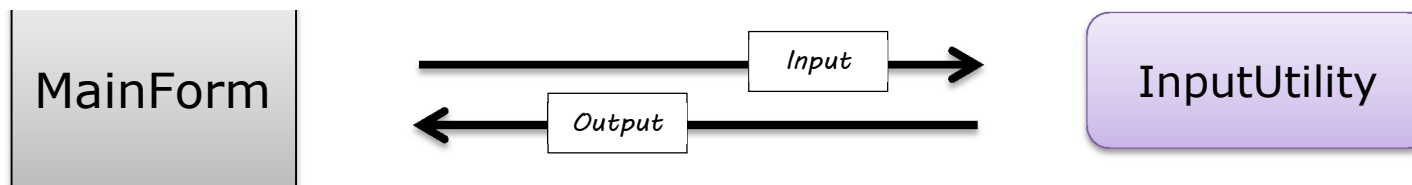
1. Overview

There are two classes to be written in this assignment:

1. **MainForm** intended to be totally responsible for handling input/output and all user-interactions.
2. **InputUtility** intended to serve as a tool (helper class) for **MainForm** for validation of input.

The **InputUtility** class is meant to be used even in the future projects and therefore should contain methods that are general and should work independent of its client classes, i.e. classes that make use of this class. In this assignment **MainForm** uses **InputUtility** and therefore depends on **InputUtility** but **InputUtility** does not depend on **MainForm**. In other words, **InputUtility** serves the **MainForm** exactly in the same way as it would serve any other class.

MainForm uses methods of InputUtility



Furthermore, because the **InputUtility** is going to contain only helper methods and offers the same operations to all its clients (other classes), we declare the methods as **static**, so the clients do not need to instantiate the class to make use of its methods. Static methods can be directly called by **ClassName.MethodName** without the need to create an instance of its class. In contrast, methods in an instance is called using **objectName.MethodName**, where **objectName** is name of the instance of an object which has been created with the keyword **new**.

MainForm calls methods of **InputUtility**, passing the required input and receiving output either through the methods' **out**-parameters, or their **return** values or both.



The class diagram in the next section illustrates the declaration of instance variables and methods in each of the two classes as a quick help. A more detailed guideline including tips and instructions are provided throughout the rest of the document.

Remember: You do not have to follow the instructions step by step and you may certainly design the GUI using your imaginations and fantasies and implement your own solution provided that:

1. you meet the requirements specified in the assignment description,
2. you maintain a good programming style specified in the quality standards outlined in a separate document available in Its L.

The above applies to all our future assignments as well.

2. Class Diagram

A class diagram is a diagram that describes classes and their relations. Here each of the two classes are shown without any association. The class diagram for the **MainForm** is given as two alternatives: the first one is for the Pass grade while the second diagram is if you are aiming at a higher grade (Pass with distinction).

Note: Most of the variables and methods that Visual Studio generate automatically are intentionally made hidden in the diagram.

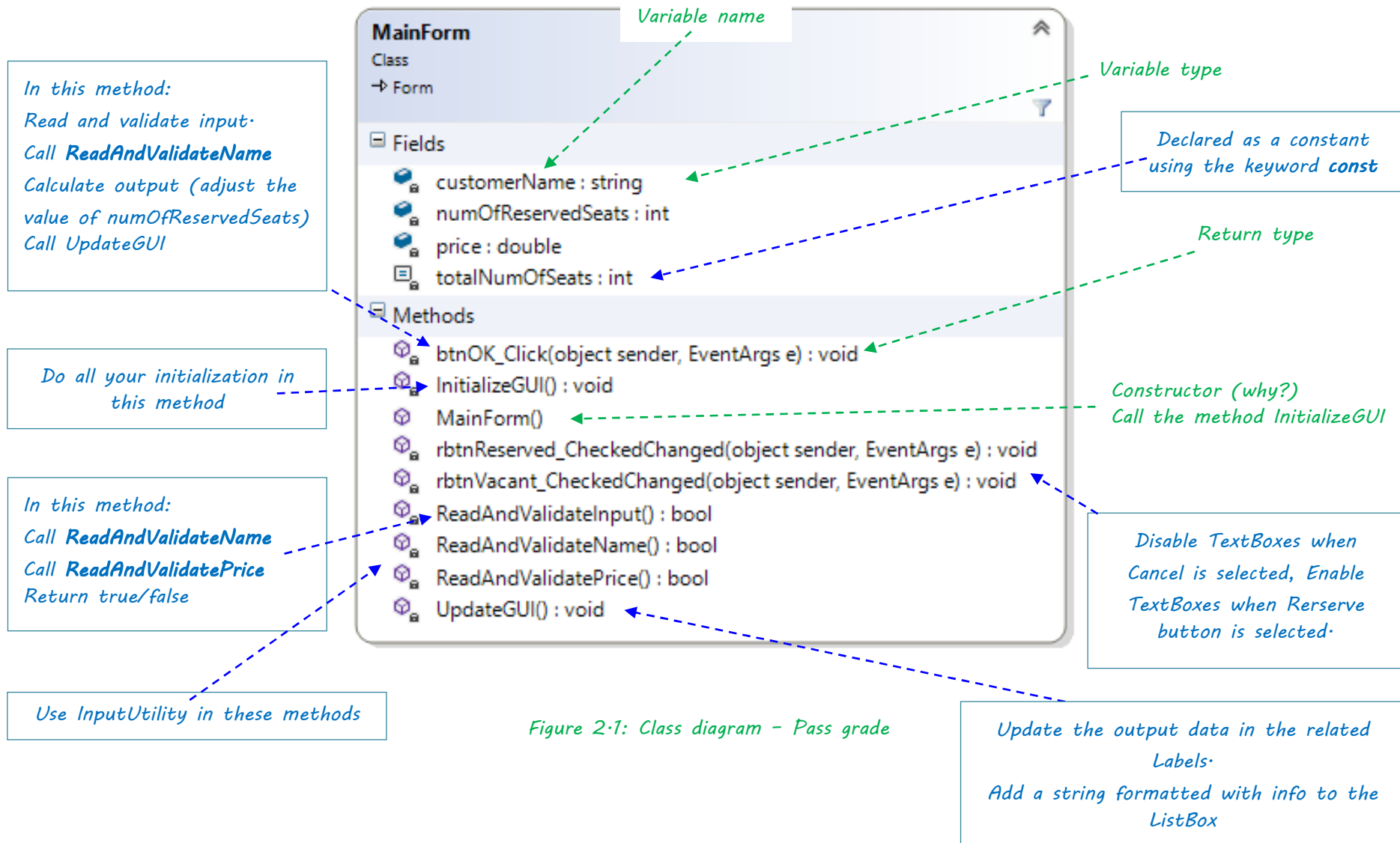
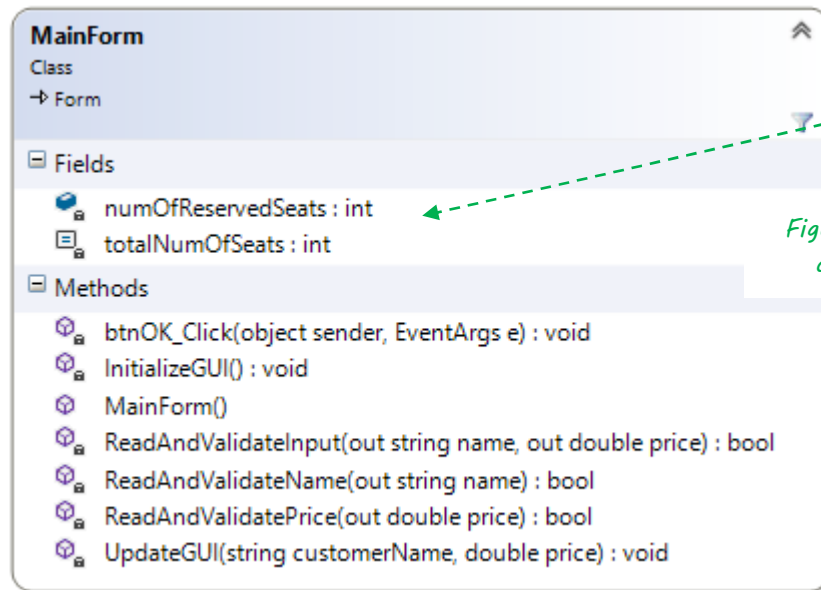
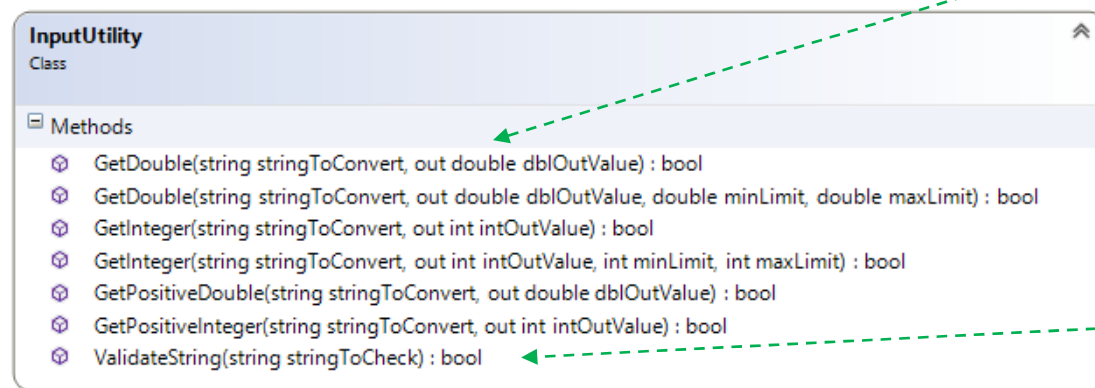


Figure 2-1: Class diagram - Pass grade



Note: no variable customer name and seat price!

Figure 2.2: MainForm Class diagram - higher grade



Method using both a pass-by-value parameter and an out-parameter

Use `double.TryParse` and return its return value.

Note: There are more methods here than are specified and required in the assignment. You can write only those which are required.

Method using only pass by value parameter

Return false if the return value of the `string.IsNullOrEmpty()` is true, and vice versa.

Figure 2.3: InputUtility Class diagram - do only the methods specified in the assignment



3. Input and output

In this assignment, we are dealing only with programming for the input. The following is the data that your program will be working with.

Input: **Reserve** or **Cancel** (bool) – to reserve a seat or cancel an already reserved seat. This information is acquired from the RadioButton's **Checked** property.

Name (string) name of the customer purchasing the ticket. The input data is acquired through the TextBox's **Text** property (**txtName.Text**).

Price (double or decimal) – price to be paid by the customer for a ticket. The user's input can be taken from the TextBox's **Text** property (**txtPrice.Text**). Because, the **Text** property of a TextBox has of the type **string** (even if the text represents a number) the string must be converted to a numerical type (double or decimal). As an example, if the user feeds in "107" which is a **string** (containing the characters 1, 0 and 7), the string must be converted to a double 107.0, or 107,0 (depending on the decimal sign set up in your Windows Regional Settings) which is a number, not a string.

Seat number (int): - The ListBox shows all the seats numbered 1 to number of seats. The ListBox has an index 0 to number of seats -1. So, no matter what type of text the lines in the ListBox show, as soon as you click on a line, the ListBox itself stores an index counted from 0. The seat numbers could start from 1000 or from 'A1'. In your code you can write as follows to detect which line is highlighted:

```
int index = lstReservations.SelectedIndex;
```

where **lstReservations** is the name of the ListBox. To check if the ListBox is highlighted, i.e. an item is clicked, you can perform the following control:

```
int index = lstReservations.SelectedIndex;  
if (index < 0)  
{  
    MessageBox.Show("Please select an item in the list!");  
    return;  
}
```

The above statement cancels further execution if the user has not highlighted an item in the ListBox.

Output: Output is the results of the calculations and manipulations of data performed by your code. If a result value is other than a string, the value is to be converted to its corresponding textual form, so it can be displayed on a Label, using the Label's **Text** property, or inside the ListBox as a string item.

Total number of seats (int) – total number of seats available for reservation. Supposing that you have a variable **totalNumOfSeats**, declared as an instance variable in the **Mainform**, for storing number of seats, the method **totalNumOfSeats.ToString()** provides a textual representation of the variable **totalNumOfSeats**.

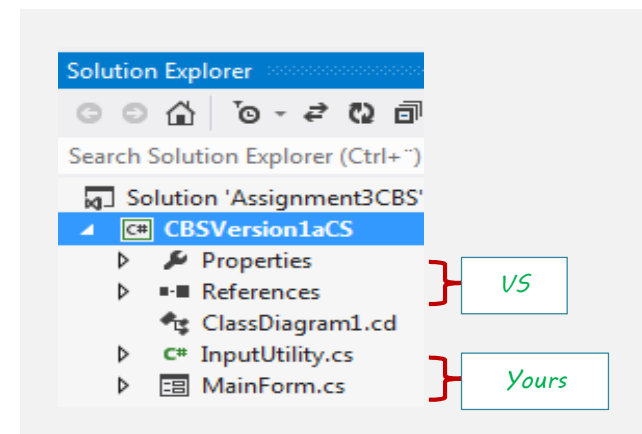
Number of reserved seats (int) – the number of seats that are reserved. Proceed as above to display the value as text in the dedicated Label.

Number of vacant seats (int) – the number of seats that are still available for reservation. This can be easily calculated by subtracting the number of reserved seats from the total number of seats. There is no need to declare an instance variable for this. When you need this info, calculated the value and save it in a local variable.

List of the seats: a list containing information about the seat number, name of the customer and the price paid if the seat is reserved. Format a string made of these values and add the string to the ListBox using the ListBox's **Items.Add** method. In this assignment we are not working with list variables and instead we make use of the ListBox control and work with one item at a time. Lists (arrays) will be covered in the next module.

4. Work plan

- 4.1 To begin with, start a new Visual C# Windows Form project in Visual Studio (VS) and start sketching the user interface (the Form, referred to as GUI hereafter). Beginning with the design of the GUI will give you an idea of which input and output data you may need to handle in your application. This helps in turn identifying some of the classes and members of the classes (instance variables, methods) that may be needed to store and handle the data.





- 4.2 VS creates a Form for you when you create a Windows Form application. The first thing you should do is to rename this class and give it a better name than the default **Form1**. Right-click on Form1 in the Project Explorer and select the sub-menu **Rename**. Give a new and suitable name (don't forget the extension cs.), say **MainForm.cs**. Doing so, VS will ask you through a **MessageBox** for permission to refactor the code to change every occurrence and reference of Form1 to **MainForm**. You should of course accept this VS's good service.
- 4.3 **The simple version (Pass grade):** Declare instance variables to store the user's input. It is the price and name of the customer that need to be parsed, and saved in variables. Some input data such as values of the radio buttons and the index of the highlighted item in the ListBox may not need to be stored as they are not the main input.
- 4.4 **The more advanced version (Higher grade):** The customer name and the seat price are to be saved in local variables (not instance variables) and these data are to be exchanged between the methods of the class through method parameters and return values. This is for purpose of working with parameterized methods using **out**-arguments.

5. Use Windows Forms Controls to design your GUI

VS contains a large number of controls (Button, TextBox, Label, PictureBox, etc) that you can use to display both input and output. Every control has a large number of properties and events. In addition, every control is used for a certain purpose. It is important that you select the right control for the right purpose. Have the following tips always in mind:

Textbox: is designed for input requiring editable textual data and remember: both alphabet and numbers as well as symbols are characters making a string. A Textbox control works with strings and it stores the text written in the box by the user at run time. The text is saved in its Property called **Text** (**textboxName.Text**). You can also assign a text value to this property in your code.

Label: is specially designed for read-only texts. It can be used as heading for other controls not having their own caption, and it can be used for output that is to be read-only such as results of a calculation. The content of this control is a string also in its **Text** property (**lblLabelName.Text**)

Differentiate between input and output controls. Textboxes should be used only for input and not for output even if it is possible to disable a textbox for editing. Do not make a TextBox act as a Label by setting its Enabled property to False.



ListBox is used for presenting a list of options (for input selection) to the user. It can also be used for displaying a list of string items (output). This control works with a list of string objects and has many useful properties, **Add**, **Remove**, **Insert**, **Clear**, **SelectedIndex**, **Items**, **Count**, etc.

RadioButton for making choices (input). This control has a property named **Checked** that gets the value true when the option is checked at run time. You can also assign a value **true** or **false** to the **Checked** property of a **RadioButton** to select or deselect the option (**rbtnReserved.Checked = true**).

Button, to start or end a process, turn on or off things, etc. (input). This control has an important event, the **Click** event, that when handled allows you to write code connected to the clicking of the button.

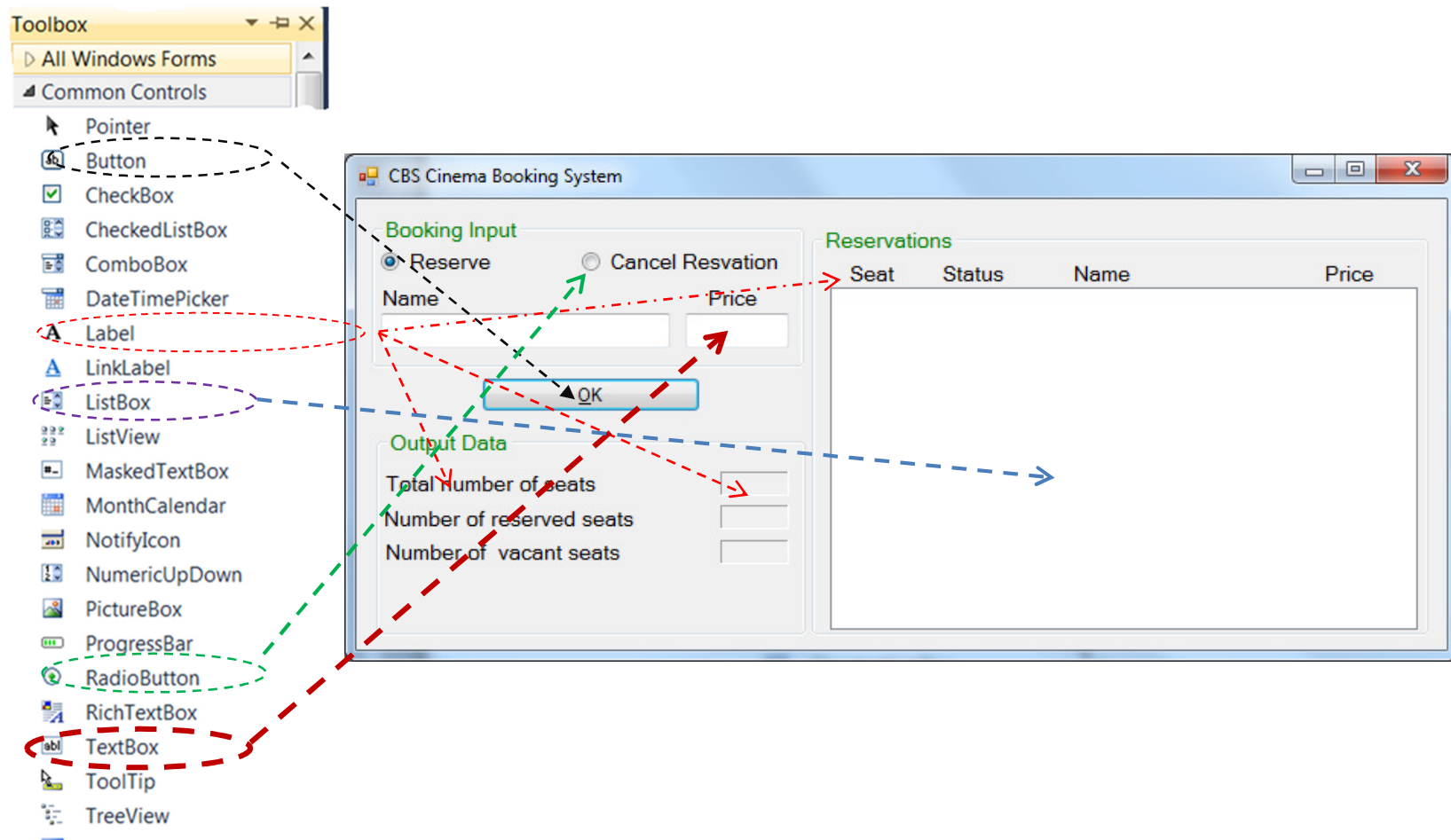
With the above introduction in mind proceed with shaping up your GUI.

5.1 Draw the GUI on your Form using the controls in the Toolbox Window in Visual Studio. Use suitable names for the controls. Generally, you should choose suitable names for everything in your application, for the solution, the project, the forms, and the classes that you write. Names like **WindowsApplication1**, **Project1**, **Textbox1** and **Label1** are default names that are not good. However, Labels used for headings and other text which are not referred to in the code (static texts), can retain their default values like **Label1**, etc, but those that will be addressed from the code must have meaningful names, like **lblTotalNumOfSeats**

5.2 The controls useful in this assignment are outlined in the table that follows. It is recommended that you begin the names of the controls with a three letter prefix as suggested in the table. (**Note**: using the Hungarian notation is not recommended anymore but keeping the prefix with controls is practical and saves times when coding).

Control	Name	Purpose
ListBox	lstSeats	Showing all vacant/reserved seats in the cinema's auditorium
GroupBox	grpInput, grpOutput	Grouping radio buttons and other input/output components.
RadioButton	rbtnReserve, rbtnCancel	Determines if you want to reserve or cancel a reservation i.e. to mark a seat reserved or vacant.
TextBox	txtName, txtPrice	Takes input from the user.
Button	btnOK	Issues a reservation / cancellation.
Label	lblTotalNumOfSeats, lblNumOfReservedSeats, etc..	Used for naming other components and for output. (Use Fixed3d border for output labels). Labels used for headings can have their default names like Label1 , etc, but those that will be addressed from code must have suitable names, like lblTotalNumOfSeats

5.3 The figure below illustrates the controls drawn on the form. You can design your GUI with a similar look, change default names as instructed above and change their properties when needed.





- 5.4 Most of the times, you may have controls that contain design-time texts, such as TextBox1 written by VS inside a TextBox. These can be of course taken care of by changing the properties of the controls in VS at design-time. However, control properties can also be changed and initialized from code. To collect all such initializations in a method is a good idea. You can write a method, let's call it **InitializeGUI** and place all the initialization code there. You can even break down this method into smaller ones as a good programming style.
- 5.5 Call the **InitializeGUI** method from the Form's constructor method (explains shortly) after the call to the method **InitializeComponent** that VS has already coded there. **InitializeComponent**, is a method that is generated automatically by VS in which VS declares the components, saves the initial values that you set (or the default values) and writes other necessary code whenever you bring changes to the Form or the controls at design time.

Let's now leave **MainForm** for a while and instead create and complete the **InputUtility** class. We will return to **MainForm** when **InputUtility** is coded and is ready for use.

6. The InputUtility class

Normally when a class needs to use another class, it creates an object of the other class and then accesses all of the object's public members. However, tools and helper classes contain very general methods and instantiation of such classes is not necessary. The same object can serve all of its clients. For this reason, when a method is declared with the keyword **static**, the method can be called directly using the class name (instead of object name) dot method name without creating an object of the class.

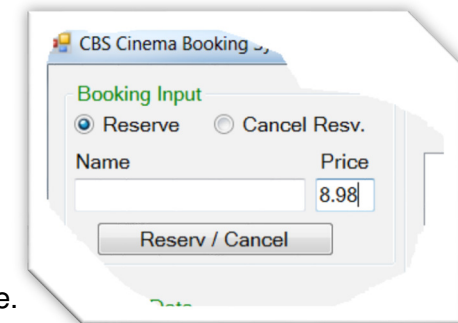
```
bool goodName = InputUtility.ValidateString(name);
```

C# provides a neat function for doing this called **TryParse** (read more at <http://msdn.microsoft.com/en-us/library/f02979c7.aspx>). Each numeric type has a **TryParse** method (in addition to Parse) for parsing textual representation of numbers into its type. The TryParse function for parsing a double from a string can be used as the example below:

```
bool goodNumber = double.TryParse(textIn, out result);
```

TryParse uses an **out** parameter for holding the converted value and returns **true** if the conversion is successful and **false** if it is not. The best feature of the **TryParse** method is that it does not cause an exception (abnormal program termination) if the string expression cannot be converted to the desired type. This is the biggest difference between the **TryParse** and the **Parse** methods attached to a type. A programmer in this way can always check the return value of the **TryParse** method to find out if the conversion has succeeded and the converted value (result in the above example) has a valid numerical format.

Consider the above example. What happens here is that the string **textIn** representing a numeric value is passed to the **TryParse** method. The string will be parsed to a numeric type and the converted value will be stored in the **result** variable. If the parsing fails the function will return false, and if it succeeds it will return true. The keyword **out** is used to tell the compiler that the variable is an output parameter. It is used as a value holder to transfer a value back to the caller method. The keyword **out** must be also used together with its variable in the caller method.



As an example assume that we have a numeric value entered in a TextBox on your form as in the figure. The input is saved in the Textbox as a string containing a sequence of characters, '8' '.' '9' '8':

```
txtPrice.Text = "8.98";
```

This is because the Property Text of a TextBox is declared as **string**, but what we need is not the string "8.98", rather a **double** value 8.98 (no quotation marks). Therefore, we must convert the contents of txtPrice.Text ("8.98") to a **double** value (8.98) and save it in a variable of **double** type

6.1 Write a method **GetInteger** with an input parameter and an output parameter. The input parameter is for passing the value of the string that is to be converted to an integer (**System.Int32**); the output parameter is a transport variable that is to contain and bring back the converted value. When using **out**-parameter, the address of a variable is passed and thus both the caller and the callee methods share the same variable (even if different names are used in the mentioned methods). Any changes to the variable in either of the methods will affect both methods.

```

/// <summary>
/// GetInteger
/// Converts a text representing an integer to an Int32 value.
/// </summary>
/// <param name="stringToConvert">string representing the integer value.
/// If the conversion is successful, this variable will have a valid value,
/// otherwise it is initiated to zero.</param>
/// <param name="intOutValue">output parameter, the converted integer value.</param>
/// <returns>true if the conversion is successful, false otherwise.</returns>
public static bool GetInteger(string stringToConvert, out int intOutValue)
{
    bool goodNumber = int.TryParse(stringToConvert, out intOutValue);
    return goodNumber;
}

```

6.2 Write a method **GetDouble** in the same manner.

You might ask why not using **TryParse** directly in the code whenever needed. You can of course do that but by coding methods in the **InputUtility**, you have at least the documentation at one place. Furthermore, you might modify the methods to make it more advanced. Finally, you have a chance to practice creating helper classes using static methods!

6.3 Write a method in the same class that validates a string for not being null or empty.

Note: the if-else block can be replaced by the following statement:

```
return !string.IsNullOrEmpty(strIn);
```

```

/// <summary>
/// Validates a string and returns true if the string is not null
/// after trimming. Does not need an out-parameter because the
/// parameter contains data, the string to be validated.
/// </summary>
/// <param name="stringToCheck">The string to check.</param>
/// <returns>True if the string is not empty after a trim and not
/// null.</returns>
public static bool ValidateString(string stringToCheck)
{
    //delete the beginning & ending spaces
    string strIn = stringToCheck.Trim();
    if (string.IsNullOrEmpty(strIn))
        return false;
    else
        return true;
    /* The above if-else block can be replaced with the following:
    return !string.IsNullOrEmpty(strIn); */
}

```

Write other methods that are specified in the assignment. Remember that you have been asked to write more methods than needed in this assignment. The other methods are for use in future projects. You can put more input handling methods now or in the future.

7. Back to MainForm

In the **MainForm**, you may need to write methods that serve different purposes as listed below:

7.1 InitializeGUI:

Write this method as a private void method and put all your initializations in this method.

7.2 ReadAndValidateInput:

In this method, call two other methods **ReadAndValidateName** and **ReadAndValidatePrice** to handle the user input given in the textboxes. This method should return true if both of the method are true and false otherwise. This method should be called from the event-handler method which is connected to the OK button (btnOK_Click).

7.3 ReadAndValidateName:

Validate the text that the user writes in the textbox used for the name of the customer. The validation should include a test of that the string is not empty after trimming the text from the spaces that the user might have added, intentionally or unintentionally, at the beginning or end of the text in the TextBox.

```
/// <summary>
/// Converts a string represented double value into a double type, and validates
/// the converted value >= 0.0.
/// </summary>
/// <param name="stringToConvert">string representing the double value.</param>
/// <param name="dblOutValue">output parameter, the converted double value.</param>
/// <returns>true if the conversion is successful and the number >= 0.0. </returns>
public static bool GetPositiveDouble(string stringToConvert, out double dblOutValue)
{
    //Try parse a string encoded double into a double data type.
    bool goodNumber = double.TryParse(stringToConvert, out dblOutValue);
    if (goodNumber)
        goodNumber = dblOutValue >= 0.0;
    return goodNumber;
}

public partial class MainForm : Form
{
    //Fields
    //A default number of seats in the cinema
    private const int totalNumOfSeats = 240;
    private int numOfReservedSeats = 0; //increase as a reservation is made

    //Input Variables to store input values given by the user
    private double price = 0.0;
    private string customerName = string.Empty;

    //No output variables needed.

    /// <summary>
    /// Default constructor
    /// Constructor is a special method that (1) has the same method name
    /// as its class (here MainForm), (2) has no return type, not even void.
    /// A constructor is automatically called whenever an object of a
    /// class is being created (with new). It is therefore a good place
    /// to do all initializations in this method.
    /// </summary>
    public MainForm()
    {
        //Visual studio's generated method
        InitializeComponent();

        //My initialization method
        InitializeGUI();
    }
}
```


7.4 ReadAndValidatePrice:

Validate so the text that user writes in the TextBox so it represents a number and that the number is ≥ 0.0 .

7.5 UpdateGUI

This method when called refreshes all output controls with current results. It can be called from different places, especially after a change of data has taken place. When a new reservation is made, the output Labels as well as the contents of the ListBox should be updated accordingly.

7.6 Double-click on the OK-button in VS to let VS prepare for you a method that will be connected to the click-event of the button. All code you write there will be automatically executed when the button is clicked at run-time. In this method you should do the following:

7.6.1 Read and control the input given by the user via the GUI and save them in variables in your program.

7.6.2 If the input is validated successfully,

- Take the necessary action and process the input data to produce output data (results).
- Update the GUI with output data.

```

/// <summary>
/// Clear the input and output controls (if needed).
/// Do other initializations, for example select one of the radio-
/// buttons as default.
/// </summary>
/// <remarks>This is to be called from the constructor, AFTER the
/// call to InitializeComponents.</remarks>
private void InitializeGUI()
{
    rbtnReserved.Checked = true; //set as default

    //clear the listbox and fill with vacant seats
    lstReservations.Items.Clear();
    for (int i = 0; i < totalNumOfSeats; i++)
    {
        string strOut = string.Format("{0,5} {1,-8} {2, -18} {3, 10:f2}",
                                     i+1, "Vacant", customerName, price);

        lstReservations.Items.Add(strOut);
    }
    //clear input controls
    txtName.Text = string.Empty;
    txtPrice.Text = string.Empty;

    //clear output controls
    lbNumberOfSeatsOutput.Text = string.Empty;
    lbNumberOfVacantOutput.Text = string.Empty;
    lbNumberOfSeatsOutput.Text = totalNumOfSeats.ToString();

    //continue with other initializations
    }
    
```

:f2 rounds up the value to 2 decimal places before conversion to text.

*{2, -18} will be replaced with the name of the customer saved in the **customerName**. The text will be left-aligned with 18 chars.*

7.6.3 If the input is invalid,

- Give an error message to the user.
- Exit the method so the user can fix the problem and try again.

The screen dump here is an example of code you can write in in the button's event-handler method. The code for **ReadAndValidateInput** method is also given below. Look at it, and make sure that you understand everything and every detail and then do the same (or similarly) in your project by yourself. Do not just copy the code lines! Notice how the methods are documented using the XML-compatible format.

Read the comments and make sure you understand the code. This will help you in solving the future assignments.

7.7 Write the methods **ReadAndValidateName** and **ReadAndValidatePrice** that are called in the above code, and make use of the proper methods from your **InputUtility** class.

```

/// <summary>
/// Event-handler method for the Click-event of the button. When the user clicks the
/// button, this method will be executed automatically.
/// Call the ReadAndValidateInput method, save its return value in a Boolean (bool)
/// variable. If the return value is true, then call the UpdateGUI method to display
/// the results.
/// </summary>
/// <param name="sender">Reference to the object that has fired the Click event
/// (the button)</param>
/// <param name="e">Contains current information about the event. </param>
/// <remarks>Sender and e are part of event delegate handling - advanced course!</remarks>
private void btnOK_Click(object sender, EventArgs e)
{
    bool inputOk = ReadAndValidateInput();

    if (inputOk)
    {
        if (rbtnReserved.Checked)
        {
            numOfReservedSeats++;
        }
        else
        {
            numOfReservedSeats--;
        }
        UpdateGUI();
    }
}

private bool ReadAndValidateInput()
{
    //call methods to validate indata
    //nameOK gets a value true or false depending on whether the
    //name is neither empty nor null.
    bool nameOK = ReadAndValidateName();

    //price OK get a value true or false depending on whether the price
    //is a valid double
    bool priceOK = ReadAndValidatePrice();

    //send true if both of the above input readings have obtained a
    //true signal, false otherwise
    return nameOK && priceOK; //true if both are true, otherwise false
}

```

8. Input reading using out-parameters (higher grade requirement)

Skip this section if you are not aiming at Pass with Distinction (ECT A, B, Swe VG)

The following functions are to be written making sure to understand and differentiate between parameters which are declared with the keyword **out** and those which neither have **out** nor **ref**!

```
private bool ReadAndValidateInput(out string name, out double price)

private bool ReadAndValidateName(out string name)

private bool ReadAndValidatePrice(out double price)

private void UpdateGUI(string customerName, double price)
```

9. Saving user input and displaying output

In this assignment, we are dealing only with reading the data from the input controls and validating the data. If the data is within the correct range, we save them in variables. For accomplishing this task, **MainForm** class needs to use the services (methods) of the **InputUtility** class - send values to **InputUtility** and receive results (output) through method parameters and return values. A method can deliver a single value back to the caller method through its return value. When a method should deliver multiple values, **out** parameters are to be used. It is up to you as a programmer to choose which value to return and which value to deliver using the out-parameters.

Fetch input and display output using proper controls:

The Radio Buttons, the two TextBoxes and the OK button in our GUI are the controls that we have designed for the purpose of input. The ListBox as well as the 3D Labels at the bottom-left part of the GUI are controls that we intend to use to show output. The ListBox meanwhile can be used for input, i.e. when the user clicks on a row in the ListBox, the selection can be used as input to the program.

Radio Button: To detect if the user has selected a radio button, use the button's **Checked** property value. For example if the user has selected the **Cancel Reservation** button, this button's **Checked** property has a value true and you can use this information to perform the required tasks.

```
if (rbtnCancel.Checked) //if the button is selected  
  
    // code
```

TextBox: The **Text** property of the control contains text that the user writes in a TextBox.

```
string name = txtName.Text; //saving input "Hommer"
```

Label: Just as the TextBox, the **Text** property contains the text displayed in the control.

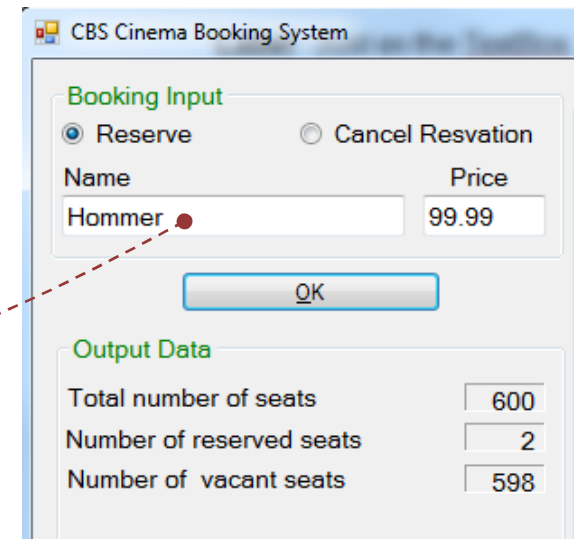
```
lbReservedSeats.Text = numOfReservedSeats.ToString(); //Output "2"
```

Button: The information we usually are interested from a button is knowing when the button is clicked. It is therefore the Click-event handler method of a button that is more important than the data it may contain. A button has also a Text as well as many other properties and methods (ForeColor, BackColor, Image, Enabled, etc.) that you can use or change.

ListBox: This control has several useful properties and methods that help receiving input and showing output.

The **SelectedIndex** property of a ListBox stores the index of the item that the user has highlighted at run-time. The index is counted from 0. If the user has highlighted/selected the first item in the ListBox, the value of the **SelectedIndex** will be 0. If the second item is selected, the value stored in the property will be 1, and so on. If the user has not selected any item, i.e. **if no item in the ListBox is highlighted** the value of **SelectedIndex** will be -1. As we are not handling any clicking of the ListBox in this assignment, we leave this discussion for the next assignment.

However, we use the ListBox as an output control for displaying a line of text with information about the reservation for cancellation of a seat. To send an item to the ListBox, you can use the **Items.Add** method as described below.



10. UpdateGUI

Every time when a new result is calculated, for example when a new seat is reserved or a reserved seat is canceled, the output data on the GUI should be refreshed and updated. To accomplish this job, it is quite practical to write a method, for example **UpdateGUI** and call it from different places in the code, where necessary. In this method, you can do the following:

- Update the output controls with current results.
- Find out which item is selected in the ListBox (index counted from 0).
- Format a string using the current values.
- Remove the item at the position = index.
- Insert the new string at the position = index.

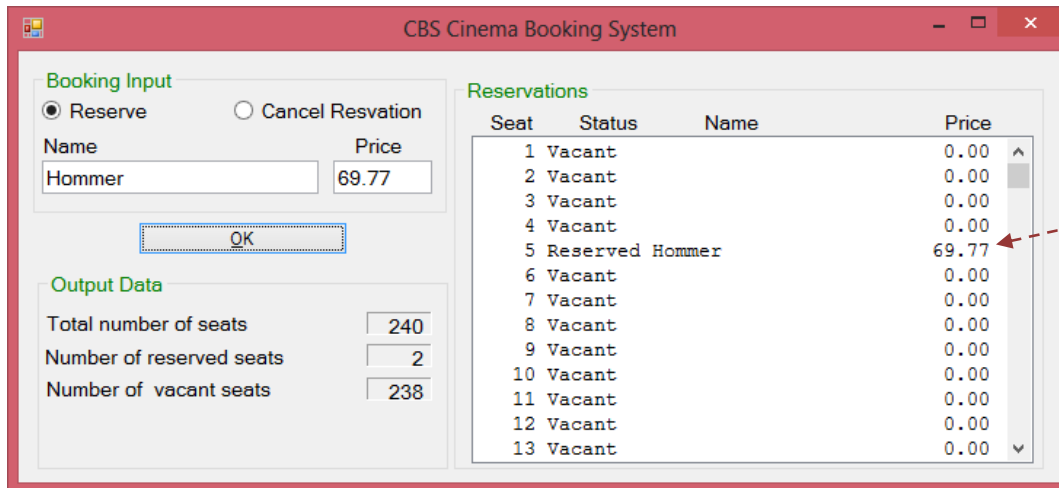
```
private void UpdateGUI()
{
    int index = lstReservations.SelectedIndex;
    if (index < 0)
    {
        MessageBox.Show("Please select an item in the list!");
        return;
    }

    string strOut = string.Empty;
    string strReserved = "Vacant";

    if (rbtnReserved.Checked)
        strReserved = "Reserved";
    else
    {
        customerName = string.Empty;
        price = 0.0;
    }

    strOut = string.Format("{0,5} {1,-8} {2, -18} {3, 10:f2}", index+1,
                           strReserved, customerName, price);
    lstReservations.Items.RemoveAt(index);
    lstReservations.Items.Insert(index, strOut);

    lbNumberOfReservedOutput.Text = numOfReservedSeats.ToString();
    lbNumberOfVacantOutput.Text = (totalNumOfSeats - numOfReservedSeats).ToString();
    lbNumberOfSeatsOutput.Text = totalNumOfSeats.ToString();
}
```



Seat	Status	Name	Price
1	Vacant		0.00
2	Vacant		0.00
3	Vacant		0.00
4	Vacant		0.00
5	Reserved	Hommer	69.77
6	Vacant		0.00
7	Vacant		0.00
8	Vacant		0.00
9	Vacant		0.00
10	Vacant		0.00
11	Vacant		0.00
12	Vacant		0.00
13	Vacant		0.00

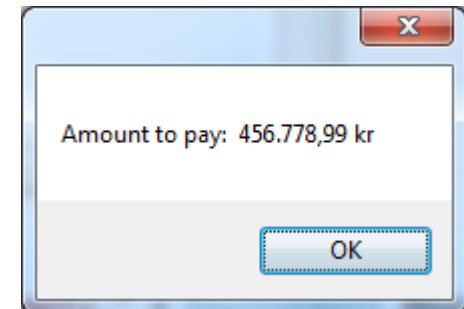
strOut saved at index = 4 (counted from 0)

11. String.Format

In the above code the string.Format is used to format a string with values separated by blank spaces. When the same format is used for all lines in the ListBox, the data will appear as columns. For this to work, you have to use a font that has a fixed for all letters, i.e. a small 'i' takes the same space on the screen or paper as a big 'W'. Tips: Change the ListBox's font to Courier New and use the string.Format as in the above code to arrange data in straight columns as illustrated in the above figure.

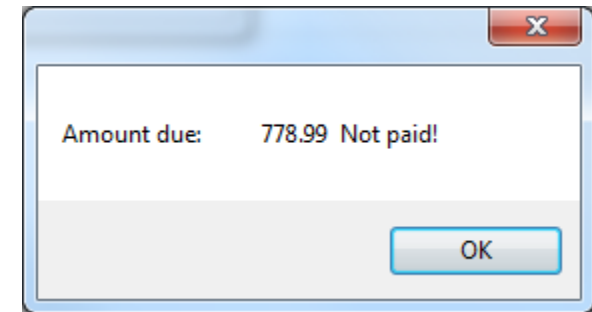
How does the string.Format works? This method is very useful to display different types of values, int, double, currency, etc as a composite text. It uses the curly brackets { } for specifying the format for each value. There are two parts in this method, a format part and a value part. The format part can contain text and formatting expression. The value part is a list of values (variables or constants). The list has zero-based index.

```
double amount = 456778.99;
string strOut = string.Format("Amount to pay: {0, 12:C}", amount);
MessageBox.Show(strOut);
```



In the format: {0, 12: C}: 0 is the index to the value in the list. The value can come from a variable or can be a constant value; here we have only one value, amount, in position 0. 12 is the width in characters within which the value is right-adjusted. If you wish to left-adjust a value, use a negative sign, for example -12. After the colon character, you can specify more options. C is used for currency and the value will be formatted using the currency settings in your operating system. To show a floating-point value rounded off to a number of decimals, for instance 2 decimals, and also left-adjust the converted text, follow the pattern below:

```
double amount = 778.990657;  
string strOut = string.Format("{0, -15}{1, 12:f2} {2, -20}",  
                             "Amount due: ", amount, "Not paid!");  
MessageBox.Show(strOut);
```



String.Format, formats a string with values separated by blank spaces before or after the text. The formatting is done within a width that is a number of characters that you specify in the format. This can be used to make the string look like column cells (no border) in a table. **String.Format** has two parts: the first part is the format, contained inside a pair of quotation marks followed by a comma and the second part containing a list of values.

In the above example, {0,-15} will be replaced by the value taken from the first position in the variable list, i.e. "Amount due: ", and is left-adjusted (because of the minus sign) inside a sub-string 15 characters long. If the text is less than 15 characters, spaces will be added to the right of the text. The {1, 12:f2} is code for formatting a floating-point value, taken from the second variable in the list (1 refers to variable number 1, counted from 0), i.e. amount; 12 specifies the width in number of characters (12 characters long) and f2 after the colon character denotes "floating-point value" to be rounded off to 2 decimal places. If the amount is as in the example 778.990657, the number is rounded off to 778.99 and converted to a sub-string containing 6 characters ("778.99"). Six blank-spaces are added to the left of the sub-string to right-adjust the text inside a 12-character space (".....778.99"). Remember that the decimal sign is also counted as a character. An important note here is to use a font with a constant width for all characters where the blank space takes the same space as a big 'W'. Such a font is Courier New. For the columns to be straight, change the font of the ListBox to Courier New!

12. What to do when the RadioButtons are clicked?

All you have to do in this part is to enable/disable the **TextBoxes**. When the Cancel Reservation button is clicked, disable the TextBoxes and Enable them when the Reserve button is clicked by the user at run-time. Double-click on the RadioButtons in VS at design-time and complete the methods that VS prepares as below:

```
private void rbtnVacant_CheckedChanged(object sender, EventArgs e)
{
    txtName.Enabled = false;
    txtPrice.Enabled = false;
}
```

```
private void rbtnReserved_CheckedChanged(object sender, EventArgs e)
{
    txtName.Enabled = true;
    txtPrice.Enabled = true;
}
```

13. To set focus to and highlight the contents of a TextBox

Being user-friendly and designing the user-interface with having the “dummy user” in mind is a sound habit and a part of making good software. To have the user’s attention to which data in the TextBoxes is invalid, you can highlight the contents of the TextBox and move the focus to that control. The TextBox control like many other controls has a method called **Focus**. It also has several methods for selecting the content of the TextBox.

A combination of the methods **SelectAll** and **Focus** highlights the content of a TextBox. This is done in the method below that is gifted to you as gratitude for reading this long but important document with patience.

Note: The highlighting works only if you do not change the focus to another control before the user receives the message. Therefore, the call to **Focus** and **SelectAll** must be placed as the last lines in your code block so the focus to the control is not lost by other operations.



```
private bool ReadAndValidateName()
{
    customerName = txtName.Text;

    if (!InputUtility.ValidateString(customerName))
    {
        // Inform the user...
        MessageBox.Show("Invalid input in the name field! Name cannot be empty," + Environment.NewLine +
            "and must have at least one character (not a blank)", "Error!",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        txtName.Focus();
        txtName.Text = " ";
        txtName.SelectAll();
        return false;
    }
    else
    {
        // Name is not empty
        return true;
    }
}
```

Only because if the TextBox is empty, there is no text to highlight!

Hopefully, this help document has given you some clue in getting an understanding of working with the Windows Form and Controls and writing event-driven code. However, there is a huge amount of information you have not seen yet, but let's take one step at a time. It should also be pointed out that Visual Studio does a lot of hard work for you behind the scenes. You can review the amount of generated code in the file **MainForm.Designer.cs**, but it is recommended not to edit this file. Other code is placed in the files in the Properties folder. Your code is saved under the folder **MainForm** in Project.

Good Luck!

Programming is fun. Never give up!

Farid Naisan,
Course Responsible and Instructor

