MALMÖ HÖGSKOLA

Programming Using C#, Basic Course

Assignment 4

Arrays

CBS

Cinema Booking System

Version 2

Mandatory

Farid Naisan
University Lecturer
Department of Computer Sciences
Malmö University, Sweden

Contents

Assignment 4: CBS Versions 2 & 3 - Arrays

1. Objectives

The main objectives of this assignment are:

- To work with one and two dimensional arrays.
- Use enumerations to group named constants.
- Use constructors to initialize an object.
- Enhance the GUI with new features and using a ComboBox.
- Exercise more with parameterized methods.

2. Description

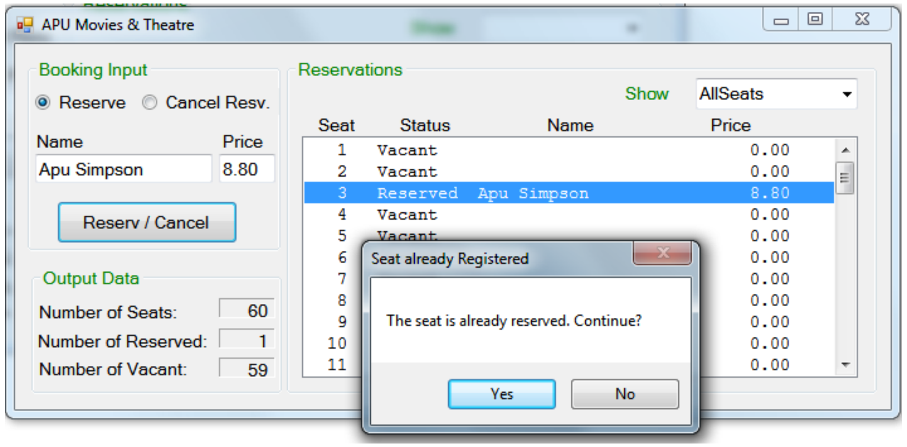
The project idea was described in the last assignment. There it was explained that you had to write a program for automating the booking of tickets for a newly established movie theater in your town. The program was to be used by the cinema's cashiers who need to register the name of the customer and the price for the seat being reserved.

In the last version, you designed the GUI and coded for handling the input. All these are to be reused (with very little changes) in this assignment. In this assignment, you are to produce a new version, version 2 of the same application by building on version1. This assignment has a mandatory part for a Pass grade, Version 2a, and an optional part, Version 2b for a Pass with Distinct grade. Then there is also another optional part, Version 3 (not graded) for those who want to do more. The main goals to be accomplished in this assignment are:

- Put functionalities so the application works properly.
- The user should be able to reserve a vacant seat.
- The user should be able to cancel an already reserved seat.

2.1 You may certainly apply your own fantasy for designing the GUI and how the cancelling and reservation features should work, provided that you take the requirements and the general quality standards set in this course (see separate document) into consideration. Otherwise follows the scenarios below:

2.1.1 List all seats when the program starts up. All seats are vacant at this time.



APU Movies & Theatre

Booking Input

(● Reserve) ○ Cancel Resv.

Name: Apu Simpson
Price: 8.80

Reservations

Show: AllSeats

```
Seat   Status                     Price
  1    Vacant                      0.00
  2    Vacant                      0.00
  3    Reserved  Apu Simpson       8.80
  4    Vacant                      0.00
  5    Vacant                      0.00
  6                                0.00
  7                                0.00
  8                                0.00
  9                                0.00
 10                                0.00
 11                                0.00
```

Output Data

Number of Seats: 60
Number of Reserved: 1
Number of Vacant: 59

Seat already Registered

The seat is already reserved. Continue?

2.1.2 The user can then click on a seat in the ListBox, write the name of the customer on the left, fill the price in the second text box, and finally click button.

2.1.3 If the input is OK, then the seat is reserved, i.e. the seat is marked as reserved in the registry (array) in the program. The output information on the GUI is then refreshed and updated.

2.2 To cancel a reservation, the user clicks on a seat in the ListBox. The seat should be already reserved. The **CancelReserv** RadioButton should have also been selected by the user, before the user clicks the button to start the process. The program should then cancel the reservation; no reading of name or price is needed. The program should register the change in the registry (array) using the highlighted index from the ListBox as the seat number.

2.3 If the **Reserve** RadioButton is selected and the user clicks the button (to reserve an already reserved seat), the program should give the user a warning message before overwriting the reservation.

2.3.1 If the user confirms the intention of overwriting by clicking the **Yes** button, the name and price should be read again from the textboxes and the registry should be updated with the new values.

2.3.2 If the user clicks the **No** button, cancel the operation (do nothing).

Some guidance and hints are provided at the end of this document. It is recommended that you pay a visit to this help section and review the paragraphs.

3. Requirements

3.1 User input must be controlled as described in the previous assignment.

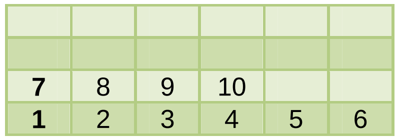
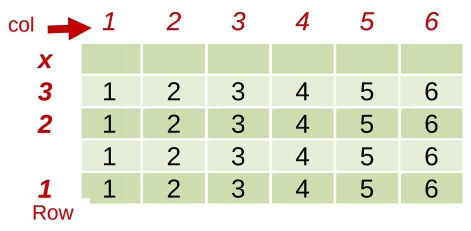
3.2 To learn using documentations available on the Internet, specially the MSDN msdn.microsoft.com , this time you are to find facts about using the type enum in C#. When checking the MSDN, select the documentation for .NET Framework 4.0. To use the help available through the different means, textbooks, MSDN, Google is something that you must get used to. This and debugging (we will train with this on another occasion) are a considerable part of a programmer’s daily life.

3.3 All user-interactions must be coded in the **MainForm**. The **SeatManager** class should not have any interaction with the user. The MainForm communicates with SeatManager through methods.

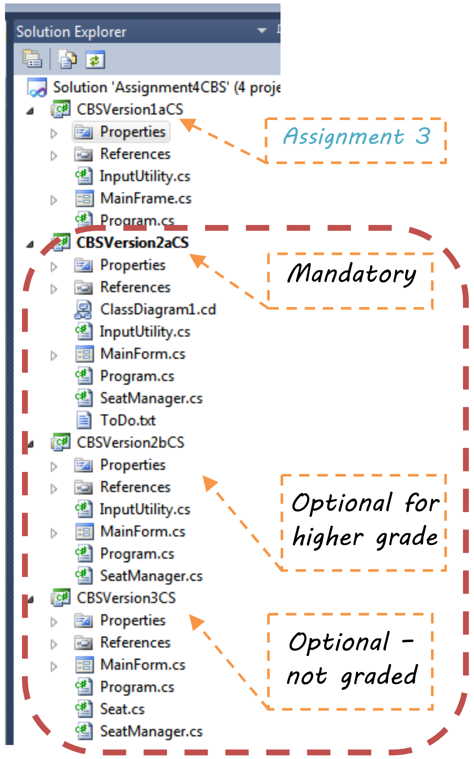
3.4 Use of collections is not allowed in this assignment. Remember that we are using one and two dimensional arrays in this assignment to exercise and learn to use them, because these are a part of the essentials in programming languages. We will be using collections a lot in the future assignments.

The table below outlines the contents of this assignment:

Table 1: Summary of the versions

Project and grading	To Do	Classes involved
CBSVersion 2a - **One-dimensional arrays** **Mandatory** Grades C (G). For a higher grade, you must also do Version 2b.	• Use the files from the last assignment (more details under Project below) . • Create a new class **SeatManager**. • Declare two one-dimensional arrays in the **SeatManager** class for storing names and prices for each seat, in a sequence (0 to number of seats-1) 	• MainForm • InputUtility • SeatManager
CBSVersion2b **Two-dimensional arrays** **Optional but required for** Grades C(G) to A(VG)	• Change the one-dimensional arrays to two-dimensional arrays in the **SeatManager**. • The seats are divided into a number of rows and all rows contain an equal number of columns. Rows are number from left to right, beginning with 1 and ending with number of columns.  col 1 2 3 4 5 6 x 3 1 2 3 4 5 6 2 1 2 3 4 5 6 1 2 3 4 5 6 1 1 2 3 4 5 6 Row • Use a two-dimensional array to store customer names for each seat, indexed with its row number and its column number, for ex. **m_nameList**[row,col] • Use another two-dimensional array for storing the prices for the seats.	• MainForm • InputUtility • SeatManager
CBSVersion3- **Array of Seat objects Optional** Not graded, but you'll receive feedback on your solution.	• Create a new class, **Seat** with fields for name, price, type of seat, etc. • Use an array of **Seat** objects in the **SeatManager** class.	• MainForm • InputUtility • SeatManager • Seat

4. The Project



Solution Explorer

Solution 'Assignment4CBS' (4 proje
- CBSVersion1aCS
  - Properties
  - References
  - InputUtility.cs
  - MainFrame.cs
  - Program.cs
- CBSVersion2aCS
  - Properties
  - References
  - ClassDiagram1.cd
  - InputUtility.cs
  - MainForm.cs
  - Program.cs
  - SeatManager.cs
  - ToDo.txt
- CBSVersion2bCS
  - Properties
  - References
  - InputUtility.cs
  - MainForm.cs
  - Program.cs
  - SeatManager.cs
- CBSVersion3CS
  - Properties
  - References
  - MainForm.cs
  - Program.cs
  - Seat.cs
  - SeatManager.cs

Copy the last assignment project to a new folder to reuse the files in this assignment. You can rename files, if needed, e.g. Assignment3.sln to Assignment4.sln. Open the solution and continue to code for this assignment.

To begin with, write the **SeatManager** class. Figure out which methods you might need to handle the booking of seats. In Version 2a, you are to use one-dimensional arrays to store values for names of the customers and the prices of the seats, and in Version2b, a two-dimensional array to replace the two one-dimensional arrays. Bring necessary changes in the MainForm class, use an object of the **SeatManager** class and makes the GUI work (more details below).

Note: You don't have to follow the pattern given here; you can apply your own solution, write your own methods and construct your classes in your own way provided that you follow the requirements outlined above and that you maintain a good structure in your coding as instructed in the document "Quality Standard and Guidelines" published in the course site (private instance variables, short methods, etc).

5. One Dimensional Arrays

5.1 The information to be saved in the program is the customer's name and the price paid for the seat. Two one-dimensional arrays, hereafter referred to as **m_nameList** and **m_priceList** are to be used, one for storing the names of the customer, and one for the prices of seats

5.1.1 The seats are arranged in a number of rows and each row has an equal number of columns (seats).

7	8	9	10		
1	2	3	4	5	6

5.1.2 Seats are sequentially numbered from 1 at the outer left of the front row ending with the last seat at the outer right of the last row.

5.1.3 It is assumed that the data related to a seat is saved in the same position in the arrays, i.e. **m_nameList** [5] and **m_priceList** [5] refer to seat number 6 (array index beginning from 0).

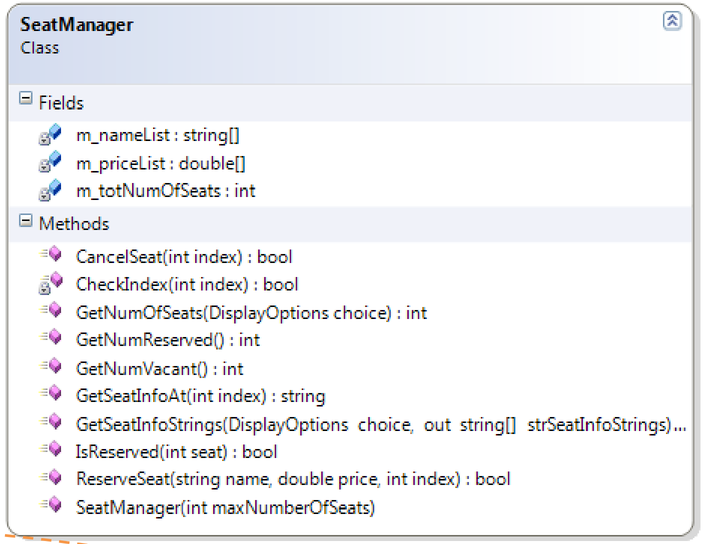
5.1.4 Declare an object of the SeatManager class as instance variable in the MainForm. Create the object in the MainForm's constructor. Using this object, list all seats in the ListBox (as vacant) at the program start.

5.1.5 When reserving a seat, the name and price should be saved in the corresponding array in the seatManager object at the position equal to the seat number -1. Why -1?

5.1.6 When canceling a reservation, the seat is to be made vacant by assigning a null value in the **m_nameList** and a 0.0 value in the **m_priceList** array.

The SeatManager Class

5.2 The Class Diagram shows a list of the fields and methods of the **SeatManager** class. The table on the next page provides some more information about the contents of this class.



SeatManager
Class

Fields
- m_nameList : string[]
- m_priceList : double[]
- m_totNumOfSeats : int

Methods
- CancelSeat(int index) : bool
- CheckIndex(int index) : bool
- GetNumOfSeats(DisplayOptions choice) : int
- GetNumReserved() : int
- GetNumVacant() : int
- GetSeatInfoAt(int index) : string
- GetSeatInfoStrings(DisplayOptions choice, out string[] strSeatInfoStrings)...
- IsReserved(int seat) : bool
- ReserveSeat(string name, double price, int index) : bool
- SeatManager(int maxNumberOfSeats)

5.3 **Fields**:

5.3.1 **m_totNumOfSeats** is the number of seats available (maximum size of the arrays). The value of this readonly variable is initialized through a constructor parameter.

5.3.2 **m_nameList**: array 0 to m_totNumOfSeats-1 is used for storing the name of the customer for whom a seat is reserved. A vacant seat has a value null.

5.3.3 **m_priceList**: array 0 to m_totNumOfSeats-1 is used for storing the prices paid for a seat, 0.0 if seat is not reserved.

5.3.4 Write a public enum **DisplayOptions** to define a number of choices as in the figure.

```
public enum DisplayOptions
{
    AllSeats,
    VacantSeats,
    ReservedSeats
}
```

5.3.5 Enumerations are very practical to group named constants. An enum is a type in C# and you instantiate them. The .NET Framework has a ready to use object called, Enum (with capital E) that operates on enums and provides many useful services.

5.4 Methods	Description	Called from

`/// <summary>`
`/// Constructor - do all your initializations here`
`/// </summary>`
`/// <param name="maxNumberOfSeats">Total number of seats</param>`
`/// <remarks></remarks>`
`public SeatManager(int maxNumberOfSeats)`
`{`
`    //only time m_totNumOfSeats can be assigned a value`
`    m_totNumOfSeats = maxNumberOfSeats;`
`    m_nameList = new string[m_totNumOfSeats];`
`    m_priceList = new double[m_totNumOfSeats];`
`}`
	Constructor with one parameter containing a value for the total number of seats. This value determines the size of the arrays. The value is provided by the object in which an instance of this class is created (MainForm).
	
	Must be public to enable other objects to instantiate this class with this constructor.
		MainForm when creating an object of the **SeatManager**.

`/// <summary>` `/// Check so the value of an index is within the array range,` `/// i.e. index >=0 and index is less than m-totNumOfSeats.` `/// </summary>` `/// <param name="index"></param>` `/// <returns></returns>` `/// <remarks></remarks>` `private bool CheckIndex(int index)`	A help function used internally in the class to check the value of an index so it is not out of range. The method needs not to be exposed to other objects and therefore it is declared as private.	Other methods in its class.
`/// <summary>` `/// Calculate the number of seats reserved.` `/// </summary>` `/// <returns>number of reserved seats</returns>` `public int GetNumReserved()`	Loop through the arrays, calculate and return the number of seats that are reserved.	**MainForm** to update output on GUI
`/// <summary>` `/// Calculate the number of seats not reserved.` `/// </summary>` `/// <returns>Number of vacant seats</returns>` `public int GetNumVacant()`	Loop through the arrays and calculate and return the number of seats that are not reserved.	**MainForm** to update output on GUI
`/// <summary>` `/// Get number of seats depending on the value of choice` `/// defined in the enum DisplayOptions.` `/// </summary>` `/// <param name="choice">a member of the enum DisplayOption</param>` `/// <returns>The number of seats for the chosen option. </returns>` `public int GetNumOfSeats(DisplayOptions choice)...`	Call one of the above two methods for the choices vacant or reserved, determined by the value of the function parameter. If the choice is AllSeats, then return the m_totNumOfSeats.	From methods in the same class or from the MainForm when needed.

`/// <summary>` `/// Adds a reservation. Save name in the nameList and the price in the priceList` `/// in the position= index` `/// </summary>` `/// <param name="index">Index of the array position.</param>` `/// <param name="name">Name of the cinema customer</param>` `/// <param name="price">Price paid for the seat.</param>` `/// <returns>True if seat is successfully reserved, False if it is already` `///  occupied</returns>` `public bool ReserveSeat(string name, double price, int index)`	This method is to be called whenever a new seat reservation is to be saved. The method receives the required input through its argument list, and saves the data in the related arrays in the SeatManager object.The index comes from the MainForms ListBox (SelectedIndex)	**MainForm** uses this service through its **SeatManager** object when the push button is clicked and the "**Reserve**" RadioButton is checked.
`/// <summary>` `/// Cancel a reservation. Assign a value Nothing in the nameList, and 0.0D in the` `/// priceList in the position = index` `/// </summary>` `/// <param name="index">Index for array position.</param>` `/// <returns>true if seat was successfukly canceled, false if the seat already is` `/// occupied.</returns>` `public bool CancelSeat(int index)`	This method is to be called whenever a reservation for a seat is to be canceled. The **SeatManager** object marks the seat in the position = index as vacant by setting the name to null and the price to 0.0d. The index comes from the MainForms ListBox (.**SelectedIndex**)	**MainForm** uses this service through its **SeatManager** object when the push button is clicked and the "**Cancel Resv**" RadioButton is checked.
`/// <summary>` `/// Returns the status for a seat in a position = index` `/// </summary>` `/// <param name="index">Index of the array position</param>` `/// <returns>A formatted string containing information about the seat,` `/// customer name, price` `/// and whether the seat is reserved or vacant.</returns>` `public string GetSeatInfoAt(int index)`	This is a method that formats an output string for a seat at position number equal to index. The method can be called for every seat using a loop inside the class. Using the method below is a better alternative.	**MainForm** or any class that wants to acquire information about a specific seat.

`/// <summary>`
`/// This method prepares an array of strings with information about all seats.`
`/// Each element is a string formatted using the GetSeatInfo function.</`
`/// </summary>`
`/// <returns> returns>`
`public int GetSeatInfoStrings(DisplayOptions  choice, out  string[] strSeatInfoStrings)`
`{`
`    strSeatInfoStrings = null;`
`    int count = GetNumOfSeats( choice);`
 	
`    if ((count <= 0))`
`       return 0;`
 	
`    string strOut = "Vacant";`
`    strSeatInfoStrings = new string[count];`
 	
`    int i = 0;  //counter for return array`
`    // is the element corresponding with the index empty`
`    for (int index = 0; index <= m_totNumOfSeats - 1; index++)`
`    {`
 	
`            //Continue`
	This method returns an array of strings, in which each element is a string that is formatted with the name of the customer and the price of the seat. This information is obtained from the arrays.
	
	The method returns an array with information about all elements of the array , for different display options.
		Mainfram can call this method and then it can use the ListBox's **AddRange** method to add and display the whole array to the ListBox – *fantastic*!.

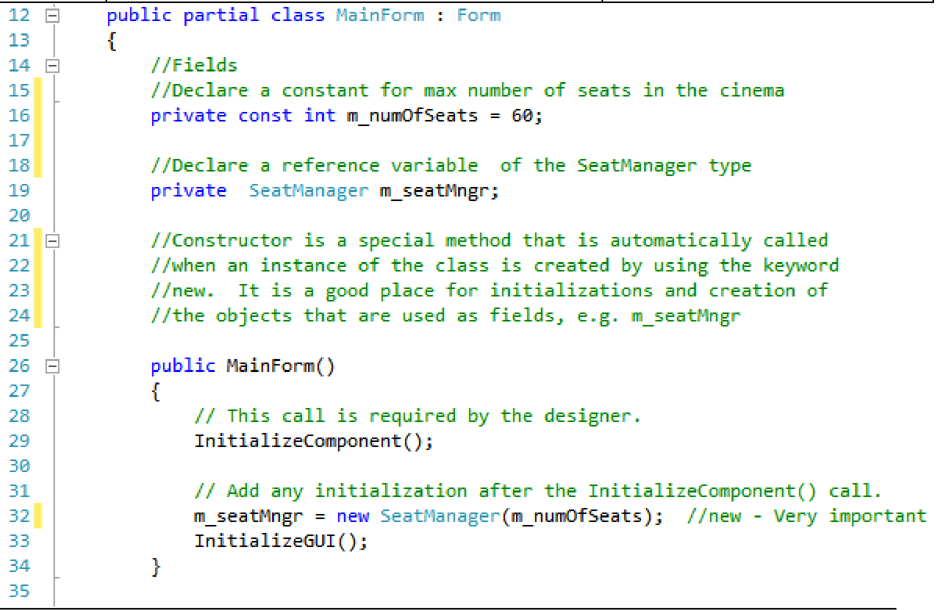
The MainForm

The **MainForm** copied from the previous version needs to undergo some changes and complementary work to accomplish the goals of this version.

Draw a ComboBox control on your GUI to give the user a choice of selecting whether to show all seats, only reserved seats or only vacant seats in the ListBox (see the GUI example given earlier in this document).

Change the **DropDownStyle** of the CombonBox to **DropDownList** to force the user to select only from given list and make the textbox part of the ComboBox read-only.

The code-clip given here at the right is to help you construct your **MainForm** object. The table that follows summarizes the major changes and revisions required to program the features of your application.



```
public partial class MainForm : Form
{
    //Fields
    //Declare a constant for max number of seats in the cinema
    private const int m_numOfSeats = 60;

    //Declare a reference variable  of the SeatManager type
    private  SeatManager m_seatMngr;

    //Constructor is a special method that is automatically called
    //when an instance of the class is created by using the keyword
    //new.  It is a good place for initializations and creation of
    //the objects that are used as fields, e.g. m_seatMngr

    public MainForm()
    {
        // This call is required by the designer.
        InitializeComponent();

        // Add any initialization after the InitializeComponent() call.
        m_seatMngr = new SeatManager(m_numOfSeats);  //new - Very important
        InitializeGUI();
    }

```

5.5 **Fields**:

5.5.1 **m_NumOfSeats** is the number of seats. This value is passed to **m_seatMngr** object as a constructor parameter at the time the object is created.

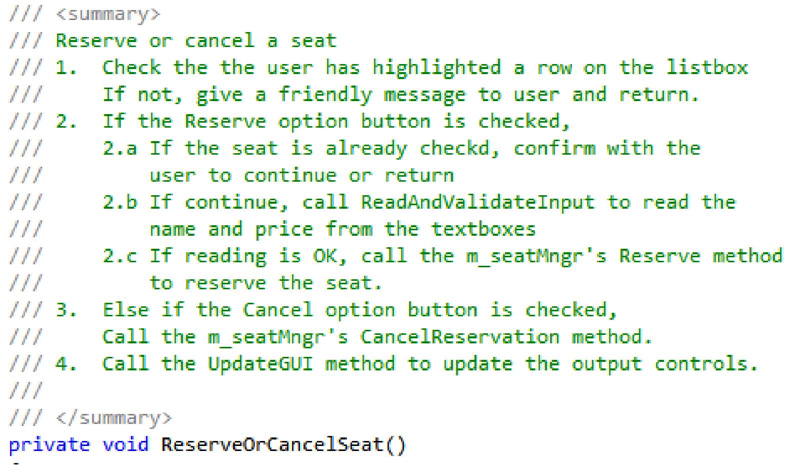
5.5.2 **m_seatMngr** points to an object of the **SeatManager** class used as a field by **MainForm**.

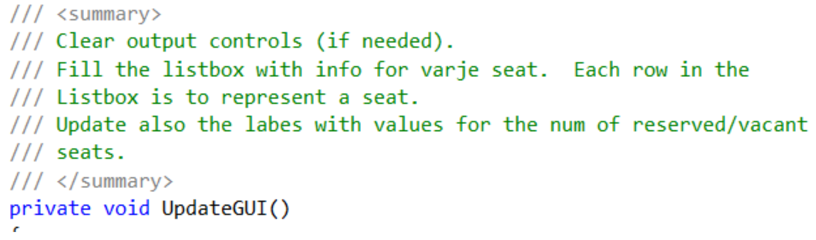
5.6 Methods	**Description**	**Called from**

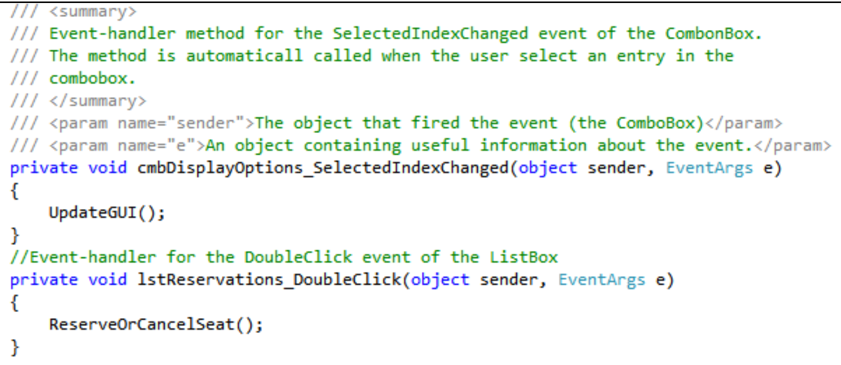
`/// <summary>`
`///  Clear the input and output controls (if needed).`
`/// Do other initializations, for example select one of the radio-`
`/// buttons as default.`
`/// Create`
`/// </summary>`
`/// <remarks>This is to be called from the constructor, AFTER the`
`/// call to InitializeComponents.</remarks>`
`private void InitializeGUI()`
	Initialize the input/out controls as in the last assignment.
	
	Fill the ComboBox with elements of the **DisplayOption** enumeration.
	
	Select the AllSeats as the default option (so the ComboBox shows this option at the program start).
	
	Call your method **UpdateGUI**, so the ListBox shows all seats, etc.
		Called from the constructor (see above code clip).

`/// <summary>`
`/// The user must highlight an item in the Listbox before a`
`/// reservation/cancelation can be performed. If an item in`
`/// the listbox is not highlighted, give an error msg to the user.`
`/// </summary>`
`/// <returns></returns>`
`/// <remarks></remarks>`
`private bool CheckSelectedIndex()`
	A utility function that checks so the value of the ListBox's **SelectedIndex** property is not negative.
	
	The value of **SelectedIndex** is the number equal to the row number (counted from 0) of the item highlighted in the ListBox. When no line in the ListBox is selected, the value of the **SelectedIndex** will be set to -1 automatically.
		Called from different places in the **MainForm** when parsing the index to the ListBox item selected by the user.

`/// <summary> ...` `private bool ReadAndValidateName(out string name)...` `/// <summary> ...` `private bool ReadAndValidatePrice(out double price)...` `/// <summary> ...` `private bool ReadAndValidateInput(out string name, out double price)...`	These are from the last assignment – no changes should be necessary	
`/// <summary>` `/// Event-handler method for the Click-event of the button. When the user` `/// clicks the button, this method will be executed automatically.` `/// If the Cancel RadioButton is checked, no need to read customer name` `/// or seatPrice.` `/// Call the method ReserveOrCancelSeat to process the reservation/cancellation` `/// of a seat.` `/// </summary>` `/// <param name="sender"></param>` `/// <param name="e"></param>` `/// <remarks>Don't worry about the sender and the parameter e at this` `/// time</remarks>` `private void btnOK_Click(System.Object sender, System.EventArgs e)` `{` `    ReserveOrCancelSeat();` `}`	Call another method responsible for the processing the reservation and cancellation of a seat	Called automatically when the button is clicked at run time.

 `/// <summary>`
`/// Reserve or cancel a seat`
`/// 1.  Check the the user has highlighted a row on the listbox`
`///     If not, give a friendly message to user and return.`
`/// 2.  If the Reserve option button is checked,`
`///     2.a If the seat is already checkd, confirm with the`
`///         user to continue or return`
`///     2.b If continue, call ReadAndValidateInput to read the`
`///         name and price from the textboxes`
`///     2.c If reading is OK, call the m_seatMngr's Reserve method`
`///         to reserve the seat.`
`/// 3.  Else if the Cancel option button is checked,`
`///     Call the m_seatMngr's CancelReservation method.`
`/// 4.  Call the UpdateGUI method to update the output controls.`
`///`
`/// </summary>`
`private void ReserveOrCancelSeat()`
	Follow the algorithm given at left or apply your own.	The method is called when the Click event of the button is fired.
		You can also handle the **DoubleClick** event of the ListBox and call this method. In this case, the user can reserve or cancel a seat by double-clicking on a row in the ListBox – it is worth trying.

 `/// <summary>`
`/// Clear output controls (if needed).`
`/// Fill the listbox with info for varje seat.  Each row in the`
`/// Listbox is to represent a seat.`
`/// Update also the labes with values for the num of reserved/vacant`
`/// seats.`
`/// </summary>`
`private void UpdateGUI()`
	Clear the ListBox and fill it again with updated information for every seat. Call the **GetSeatInfoStrings()** of the m_seatMngr to receive an array of strings. Add the array to the ListBox, using its AddRange method. Update also the output controls at the left side.	Call this method in the **MainForm** any time an update of the output is required, especially when a new reservation or a cancelation takes place, as exemplified in the next row of this table.

 `/// <summary>`
`/// Event-handler method for the SelectedIndexChanged event of the CombonBox.`
`/// The method is automaticall called when the user select an entry in the`
`/// combobox.`
`/// </summary>`
`/// <param name="sender">The object that fired the event (the ComboBox)</param>`
`/// <param name="e">An object containing useful information about the event.</param>`
`private void cmbDisplayOptions_SelectedIndexChanged(object sender, EventArgs e)`
`{`
`    UpdateGUI();`
`}`
`//Event-handler for the DoubleClick event of the ListBox`
`private void lstReservations_DoubleClick(object sender, EventArgs e)`
`{`
`    ReserveOrCancelSeat();`
`}`
	Double-click on the ComboBox and call the UpdateGUI method in the event-handler method.
	Handle the SelectedIndexChanged event of the ListBox.
	- Select the ListBox. Select the
	- icon ⚡ to view all events for a ListBox.	

Remember:

- Make sure to always check the value of the **SelectedIndex**, because as soon as the ListBox loses focus, the value of the ListBox object's **SelectedIndex** is set to -1 automatically.

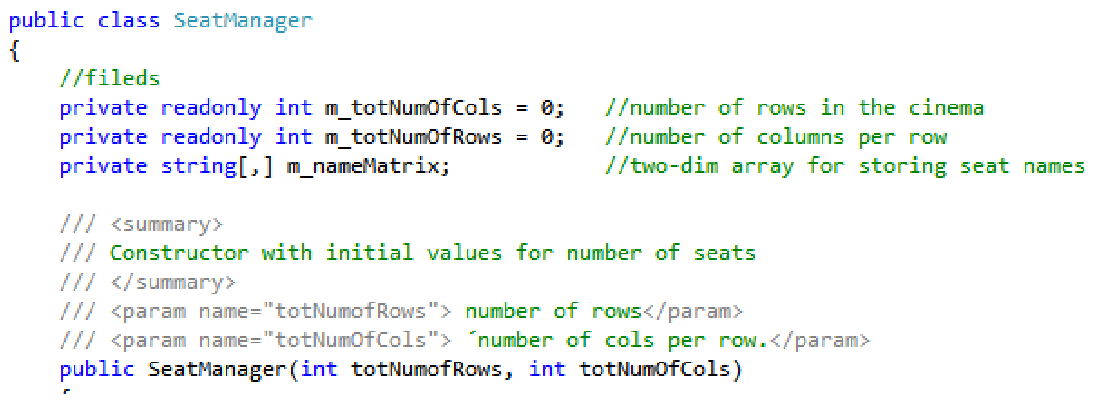
6. CBSVersion2b: Two-Dimensional Arrays

In this version seats are divided into a number of rows and columns. A seat is identified by a double index (row, col); for example m_nameMatrix[4, 3] refers to the seat on row number 5 and column number 4 (seats are numbered in the cinema from 1, but in our arrays from 0.

Add a new project in the solution; name it CBSVersion2b. Delete the Form1.cs file. Copy the classes **MainForm**, **InputUtility** and **SeatManager** from the previous version (drag and drop in the Solution Explorer). It is assumed that all rows have the same number of seats

The SeatManager Class

6.1 Replace the two one-dimensional arrays (m_nameList and m_priceList) by two two-dimensional arrays (m_nameMatrix and m_priceMatrix). **To limit the amount of your programming, it is accepted to implement only m_nameMatrix,** and let all seats have a constant ticket price as illustrated in the run example. Whether you implement both matrices or only one of them will have not affect your grade.



```
public class SeatManager
{
    //fileds
    private readonly int m_totNumOfCols = 0;   //number of rows in the cinema
    private readonly int m_totNumOfRows = 0;   //number of columns per row
    private string[,] m_nameMatrix;            //two-dim array for storing seat names

    /// <summary>
    /// Constructor with initial values for number of seats
    /// </summary>
    /// <param name="totNumofRows"> number of rows</param>
    /// <param name="totNumOfCols"> ´number of cols per row.</param>
    public SeatManager(int totNumofRows, int totNumOfCols)
```

6.2 Write a constructor with two parameters, one for input of total number of rows and one for input of the total number of columns.

6.3 The total number of rows and the total number of columns are set by the client objects (MainForm here) when calling the constructor as explained earlier.

6.4 Change the methods and do other necessary coding so your **SeatManager** class is now working with two-dimensional arrays instead of one-dimensional ones. If you wish to get some help, some code snippets are provided at the end of this document.

6.5 The probably hardest task in this version is to map en entry in the list box to an index (row, col) in your matrices. Therefore, it is recommended that you write the following methods.

6.5.1 A method that returns an index to an element at the position (row, col) that corresponds to a position in a one-dimensional presentation of the matrix. It requires a simple algorithm (solution given at the end of this document).

6.5.2 A method that returns an index to an element saved in the matrix at the position (row, col) that corresponds to a given index in a one-dimensional array (ListBox), (solution given at the end of this document).

The MainForm Class

6.6 The GUI should now also manage the input for both rows and columns and ListBox should display these too.

6.7 Bring other necessary changes in the code file to get things work.

7. CBSVersion3 – Array of Objects

This version is optional and is presented as extra exercise. It is desired by the cinema owner to store more data about a seat,

- Seat category (Business Class, Economy Class, Handicap Seat, Staff Seat) etc).
- The phone number of a customer.
- The first name and the last name separately.
- Rows may have different number of seats.

As you may have noticed from the descriptions so far, each time we have to add and handle a new type of data for a seat, there comes a lot coding, if we are determined to continue using one or two dimensional arrays. To create more arrays is definitely not an effective way.

A good solution is to define a class **Seat** and encpsulate every data desired about a seat, and write methods to handle all operations about a seat. In the **SeatManager** class, you can then declare a fixed-size array (as in the previous versions) with element of the Seat class. Fixed-sized arrays are sometimes referred to as static array, meaning that the size of the array is determined at compile time. We will be utilizing mostly dynamic arrays of objects in the future. During the rest of this course as well as in advanced courses (and also in our daily programmer lives), we will be using collections, which are advanced list types but are easy to work with. .NET supports several collection types.

7.1 Create a new project in your solution and name it CBSVersion3. Copy files from Version 2, or create new ones and then copy code after your needs.

The MainForm Class

Design your GUI using your own fantasy. Write the **Seat** and **SeatManager** classes first and then come back to the MainForm and write code so everything works the way expected.

The Seat Class

7.2 Create a new class **Seat** and complete it with fields for the above data and methods necessary to perform its tasks.

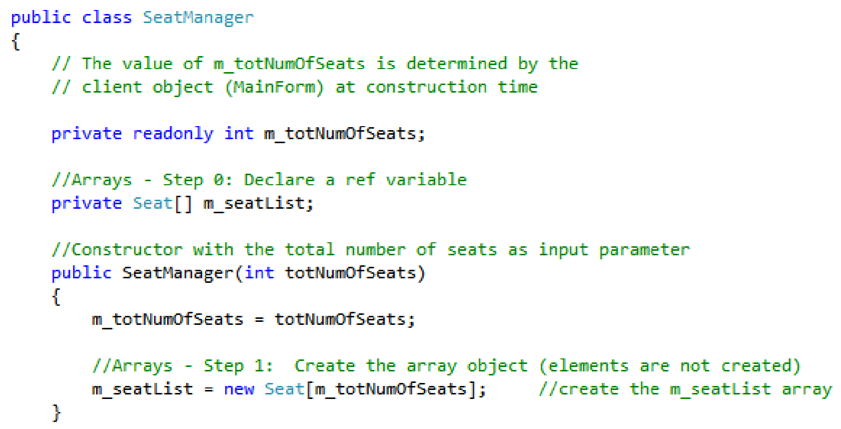
7.3 It is both practical and a good style to use Enums for defining related constants. The ones demonstrated in the figure are declared public so they could be easily be used by other classes as well – by the way an Enum is a type like a class.

The SeatManager Class

7.4 Create a new class (not much of the prevision version may be useful).

Important: In the code, shown here, the array is created, but as you may recall from the lessons, there are always two steps with arrays: (1) creating the array object (ref variable) and (2) creating each of the elements. The m_seatList in the code example is a reference variable to an array that is going to contain elements of the Seat type, i.e. Seat objects. , The variable is initiated by the compiler to a value of null.

Any reference to en element of the array that is not yet created will cause an exception and an abnormal termination of the program. You must create the elements of an array of objects as soon as you need to save an object in the array. You can either call a method that creates all the elements with default values or create and delete every element when needed. The former alternative is mentioned only because of using a fixed sized array; otherwise, the latter alternative is to prefer.



```
public class SeatManager
{
    // The value of m_totNumOfSeats is determined by the
    // client object (MainForm) at construction time

    private readonly int m_totNumOfSeats;

    //Arrays - Step 0: Declare a ref variable
    private Seat[] m_seatList;

    //Constructor with the total number of seats as input parameter
    public SeatManager(int totNumOfSeats)
    {
        m_totNumOfSeats = totNumOfSeats;

        //Arrays - Step 1:  Create the array object (elements are not created)
        m_seatList = new Seat[m_totNumOfSeats];     //create the m_seatList array
    }
```

You can create an element with the keyword new

`m_seatList[index] = new Seat();` and delete an element by letting it point to null.

`m_seatList[index] = null;`.

7.5 Write the following methods:

- AddnewSeat (arguments)
- DeleteSeat (index)
- ChangeSeat(index, arguments)
- GetSeat(index)
- GetSeatInfo(index)

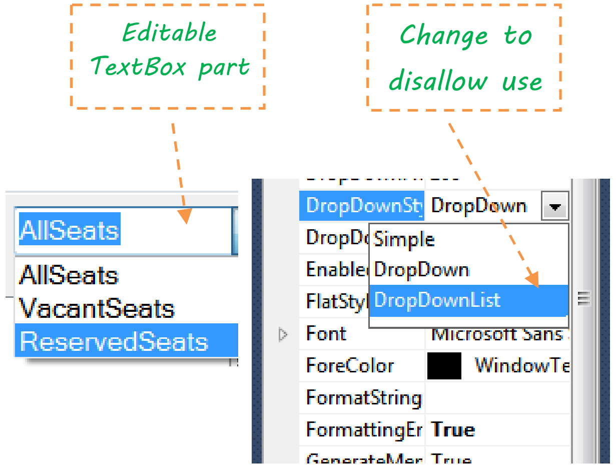
You may need to write other methods too.

```
private void TestArrays()
{
    //Arrays - Step2:  Create each of the elements (where and when you need them)
    //Create the first objectAn example only
    m_seatList[0] = new Seat();
    //first element created

    //Delete the first element
    m_seatList[0] = null;

    //Delete the whole array
    m_seatList = null;

    //resize the array
    m_seatList = new Seat[m_totNumOfSeats * 2];

}
```

8. Help and Guidance

Some hints for Version 2a:

ListBoxes and ComboBoxes have many properties and methods that are common to both. The properties **Items.Add, Items.AddRange, SelectedIndex** and **Items.Clear** are some examples. The main difference between a ComboBox and ListBox is that a ComboBox has also a textbox part which can be edited by the user. The user can also select an option from the list. The text content is saved in the ListBox's Text property (cmbDisplayOption.Text). If you don't code to work with this feature of the ComboBox, you should not allow the user to edit the textbox part of the ComboBox. To accomplish this, change the **DropDownStyle** of the CombonBox to **DropDownList** using the Properties window in the VS designer.



To fill the elements of an enum to a ComboBox, you can do as follows:

```
//Add the DisplayOptions to the ComboBox
cmbDisplayOptions.Items.AddRange(Enum.GetNames(typeof(DisplayOptions)));
```

The ComboBox's SelectedIndex get a value 0 or higher when an option in the list is selected by the user (just as with ListBoxes). When no entry is selected, the SelectedIndex has a value of -1. It is important to always check that the SelectedIndex is 0 or larger before using the index.

To select the first entry (option) in the ComboBox as default, set

```
cmbDisplayOptions.SelectedIndex = 0;
```

To convert from the SelectedIndex (which an integer) to a value of an enum type, you can use the following example:

```
(DisplayOptions)cmbDisplayOptions.SelectedIndex
```

and to convert from enum-element to an int you can do as follows:

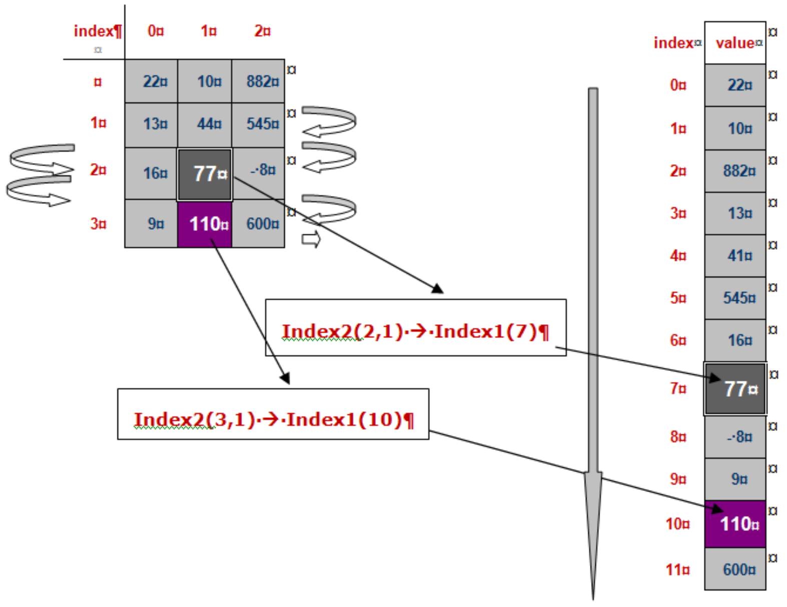
```
cmbDisplayOptions.SelectedIndex = (int)DisplayOptions.AllSeats;
```

Help with Version 2b:

A note on the choice of matrices vs one-dimensional arrays

As a programmer you can almost never avoid lists of data of same type. Databases are of course one of the best solutions, but it is not always you work directly with databases

In scientific formulas, there occur matrices representing some type of data, and it is of course more natural to program the matrices as they are, but even in this case you can save the data in a lineaar array and refer to it with a row-column index. You can write a litte function that maps a two-dimensional array index to a position in a one-dimensional array to access the same value. This is best explained by the drawing presented here.



The figur shows that if a matrix (two-dimensional array) is rolled out to a one-dimensional list, a value saved for example in the position (3, 1) has a position 10 in the one-dimensional array. Try to figure out a little algorithm to change index from a matrix system to one-dimensional array.

The code below can be used for mapping between elements a two-dimensional array and a one dimensional array.

```
/// <summary>
/// Converts matrix row and col values into single vector index value
/// </summary>
/// <remarks>This method is O(1)</remarks>
/// <param name="row">Row</param>
/// <param name="col">Column</param>
/// <returns>Index value</returns>
public int MatrixIndexToVextorIndex(int row, int col)
{
    row = row < 0 ? 0 : row;
    col = col < 0 ? 0 : col;
    return ((row + 1) * (m_totNumOfCols)) - ((m_totNumOfCols) - col);
}
```

```
/// <summary>
/// Determines the index in the matrix (row, col) that corresponds to  a given
/// index in a one-dim array (listbox). This method actually is a reverse process of
/// the method MatrixIndexToVectorIndex (see above).  The paramter row contains
/// the input, i.e. index of the element in a one-dim array. The results (row and col)
/// are saved and sent back to the caller via the ref variables row, col.
/// </summary>
/// <param name="row">Input and output parameter.</param>
/// <param name="col">Output parameter.</param>
/// <remarks></remarks>
public void VectorIndexToMatrixIndex(ref int row, ref int col)
{
    int vectorRow = row;   //row in a one dimensional array

    row = (int)Math.Ceiling((double)(vectorRow / m_totNumOfCols));  //row in the matrix
    col = vectorRow % m_totNumOfCols; //col in the matrix
}
```

Help with Version 3:

Consider the following code and continue on your own. The enums can be saved in separate files just as classes. An enum is a type in C#

```
public enum SeatCategory
{
    Business,
    Economy,
    Handicap,
    Staff
    //some seats are reserved for staff
}

public enum SeatStatus
{
    Reserved,
    Vacant,
    Unavailable      //seat not available due to repartions
}
```

```
public class Seat
{
    //which row the seat belongs to
    private int m_row;
    //which column the seat belongs to
    private int m_col;

    //first name of the customer (should be a part of a new class Customer)
    private string m_firstName;
    //last name of the customer (should be a part of a new class Customer)
    private string m_lastName;

    //Seat category, Business, Economy, etc.
    private SeatCategory m_category;

    //Seat status whether the seat is reserved, vacant, etc.
    private SeatStatus m_status;

    //put more fields if you wish
    //continue with constructors
    //continue with properties and methods

}
```

Multidimensional arrays vs one-dimensional array of objects

Consider a three-dimensional array, **`seats[5,12, 50]`** which for example represents the seat number 12 in row 5 that costs 50. How informative is this, really? Not at all, I would say. Consider now the construction given in the previous page and the SeatManager below (only a part of the class)

We have now an array of objects indexed from zero to somethting, say 499. **m_seatList[10].m_row** gives you the row number for the seat saved in position 10 and **m_seatList[10].m_Category** gives the type of a seat. . Other dimensions have readable names like row, col, price, etc, so you don't have to guess which dimension represents which information.

I'm quite sure that you agree with me that the this code speaks for itself. In addition to having many dimensions (count each member, row, col, price, etc of the **Seat** class as one dimension), It is quite easy to work with and maintain the such a structure.

In most cases even an array with one dimension, can be constructed as an array of objects in which the object belongs to a class with only field. This will give you a more maintainable structure.

```
public class SeatManager
{
    // The value of m_totNumOfSeats is determined by the
    // client object (MainForm) at construction time

    private readonly int m_totNumOfSeats;

    //Arrays - Step 0: Declare a ref variable
    private Seat[] m_seatList;

    //Constructor with the total number of seats as input parameter
    public SeatManager(int totNumOfSeats)
    {
        m_totNumOfSeats = totNumOfSeats;

        //Arrays - Step 1:  Create the array object (elements are not created)
        m_seatList = new Seat[m_totNumOfSeats];     //create the m_seatList array
    }
```

As far as two- and multidimensional arrays are concerned, I have through my many years of developing engineering programs have experienced that using arrays with two or more dimensions are not very practical, from a developer's point of view. It is much easier to make use of one-dimensional arrays of objects.

To summarize, avoid using arrays that are more than one-dimensional, unless there are special reasons. Make a class and use a one-dimensional array with elements of the class. We are going to work this way in the coming assignments.

Good Luck!

Programming is fun. Never give up. Ask for help!

Farid Naisan
Course Responsible and Instructor