Programming Using C#, Basic Course

Assignment 4

One and two dimensional arrays

Help and guidance

Apu Movies and Theatre

Cinema Booking System

Version 2

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

Contents

Help on Assignment 4: CBS Versions 2 & 3 - Arrays

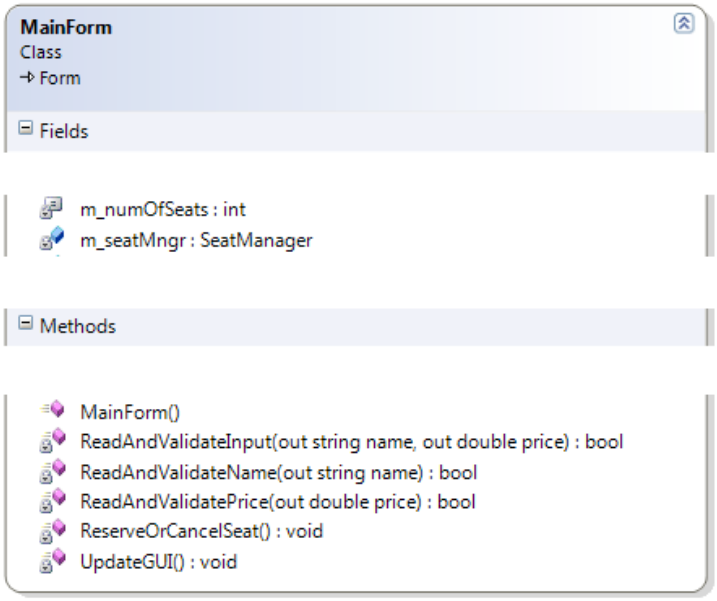
1. Objectives

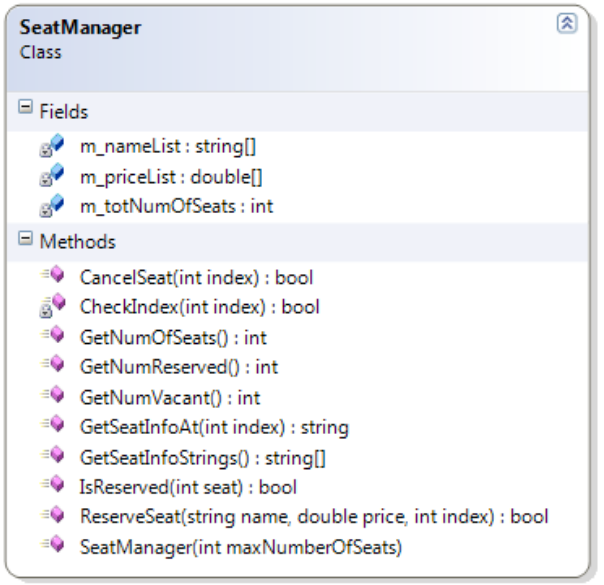
The objectives of this document is to give some help, hints and guidance to assist you accomplish the assignment. We begin with the part that is required for a passing grade. Some instructions will also be provided for those who are aiming at higher grade and would like to do the extra requirements.

Version 2a: One-dimensional array

2. Class diagram

The left part of the diagram shows a list of the methods that you might need to write. It also shows the instance variables needed. m_numOfSeats is a private constant that you can initiate to any number you wish, for ex 60 or 240 and this will determine the size of the array to be created in the SeatManager object.





3. SeatManager

3.1 Create a class **SeatManager** and save it as SeatManager.cs.

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

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

```
public class SeatManager
{
    //Input variables
    //total number of seats, initialized in the constructor
    private readonly int m_totNumOfSeats;
    //array storing customer names
    private string[] m_nameList = null;
    //array storing seat prices
    private double[] m_priceList = null;
```

3.4 **Fields**:

3.4.1 **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. A readonly type can only be assigned a value when it is initiated, either at declaration or in a constructor. The first alternative has the same effect as constant variable, while the latter allows the client objects (other objects that use this object) to provide an initial value. In this assignment we let MainForm set the value through the constructor call.

3.4.2 **m_nameList**: array indexed from 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.

3.4.3 **m_priceList**: array indexed from 0 to m_totNumOfSeats-1 is used for storing the prices paid for a seat; it has a value 0.0 if seat is not reserved.

3.4.4 In the code example here, the arrays are declared but not created. Remember that arrays are objects and they must be created using the keyword new. There are at least two places you perform the declaration: (1) by initiation on the same lines the declaration:

Private string[] m_nameList = new string[m_totNumOfSeats];

(2) in the constructor as shown in the code clip down here.

3.4.5 Write a constructor with one parameter to expect an initial value for the variable m_totNumOfSeats. Because the m_totNumOfSeats is declared readonly in the code, it can only get a value in the constructor or at declaration. As we want the client objects (MainForm) to decide about the size of the array, we choose the latter option, i.e. initiating m_totNumOfSeats in the constructor.

```
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];
}
```

3.5 The table below lists the methods in the SeatManager class as listed in the class diagram presented earlier and describes them with some more details. You don't have to do the same methods if you prefer to redesign the solution. The same is true the methods in the MainForm.

3.6 Methods	Description

`/// <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.
	
	When **MainForm** creates an object of this class, it also sends a value, for ex 240 to this class via this constructor. The value is then saved in the m_**totNumOfSeats** in the object, and the array is created with this size
	
	.

`/// <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. It returns true if the parameter index is >= 0 and < the total number of seats. (The last position in an array is one less than the size of the array). The outer boundary of the array must be less than **m_nameList.Length**.
	
	The method needs not to be exposed to other objects and therefore it is declared as private.

`/// <summary>`
`/// Calculate the number of seats reserved.`
`/// </summary>`
`/// <returns>number of reserved seats</returns>`
`public int GetNumReserved()`
	Calculates the total number of reserved seats. Run a loop through the array, calculate and return the number of seats that are reserved. A seat is reserved if the value of the element in the **m_nameList** in a position not **null** or empty. You can use the `String.IsNullOrEmpty method.`The **GetNumReserved** method is called by the **MainForm** to update output on GUI.

`/// <summary>`
`/// Calculate the number of seats not reserved.`
`/// </summary>`
`/// <returns>Number of vacant seats</returns>`
`public int GetNumVacant()`
	As above but now check so the value saved in an element in m_nameList is **null** or empty.
	
	This method is called by the MainForm to update output on GUI.

`/// <summary>`
`/// Get total number of seats. MainFrame has already`
`/// this number. It is good to have this function`
`/// anyway.`
`/// </summary>`
`/// <returns>The total number of seats </returns>`
`public int GetNumOfSeats()`
`{`
`  return m_totNumOfSeats;`
`}`
	Return the **m_totNumOfSeats** as this is a private instance variable.
	
	MainForm in this assignment sets this values but it is good to have the function, so you are prepared to have this data from the user as input in the future.

`/// <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, i.e when the **OK** button is pressed in the **MainForm**.
	
	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 **MainForm's** ListBox (SelectedIndex)

`/// <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 value of the element in **m_nameList** to null and the price in the **m_priceList** to 0.0d. The index comes from the **MainForms** ListBox (.**SelectedIndex**)

```
/// <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 method can be used by **MainForm** to acquire data to display item by item in the ListBox.

This is a method that formats and returns an output string for a seat at the position number that is equal to index. The method can be called for every seat using a loop inside the class or in the **MainForm**. The latter alternative is more object-oriented and is to prefer. The method below uses this pattern.

```
/// <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 string[] GetSeatInfoStrings()
{
    int count = GetNumOfSeats();

    if ((count <= 0))
        return null;

    string[] strSeatInfoStrings = new string[count];

    // is the element corresponding with the index empty
    for (int index = 0; index <= m_totNumOfSeats - 1; index++)
    {
        strSeatInfoStrings[index] = GetSeatInfoAt(index);
    }
    return strSeatInfoStrings;
}
```

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 name and price arrays.

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*!..

You have all code for this method, but you are expected that you take a moment, review all lines and make sure that you understand the code wholly.

4. The MainForm

The **MainForm** copied from the previous version (last assignment) needs to undergo some

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

4.1 **Fields**:

4.1.1 **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.

4.1.2 **m_seatMngr** is an object of the **SeatManager** class and is used as a field by **MainForm** (aggregation) to hold data for all seats in the movie.

```
12  public partial class MainForm : Form
13  {
14      //Fields
15      //Declare a constant for max number of seats in the cinema
16      private const int m_numOfSeats = 60;
17
18      //Declare a reference variable  of the SeatManager type
19      private  SeatManager m_seatMngr;
20
21      //Constructor is a special method that is automatically called
22      //when an instance of the class is created by using the keyword
23      //new.  It is a good place for initializations and creation of
24      //the objects that are used as fields, e.g. m_seatMngr
25
26      public MainForm()
27      {
28          // This call is required by the designer.
29          InitializeComponent();
30
31          // Add any initialization after the InitializeComponent() call.
32          m_seatMngr = new SeatManager(m_numOfSeats);  //new - Very important
33          InitializeGUI();
34      }
35
```

4.2 **Methods**	**Description**

`/// <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
	
	Call your method **UpdateGUI**, so the ListBox shows all seats, and updates other output data in the controls (Labels) designed for them, whenever a seat is reserved or a reservation is cancelled.
	
	It is usually a good idea to have a method of this type in your forms to call whenever a change in input/output requires an update of the GUI.

`/// <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 whether an item on the ListBox is highlighted by the user (or from the program code). It returns true if an item is highlighted and false otherwise. It informs the user through a MessageBox to select an item before a reservation or cancellation can take place.
	
	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.

`/// <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 methods 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();`
`}`
	The logics for reserving or cancelling a seat are placed in this method in a separate method. The method is called here.

```
/// <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()
```

This method is an important one as it starts the main task of the program, i.e. to reserve or cancel a seat. The algorithm for this method is given as XML comments above the method signature .Follow the algorithm given at left, or apply your own.

```
/// <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 refill it with updated information for every seat taken from the object of the SeatManager, m_SeatMngr.

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.

Version 2b: Two-dimensional array

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 displayed from 1 as in the theatre, but in our arrays they are numbered from 0).

5. The SeatManager Class

5.1 Declare two two-dimensional arrays (m_nameMatrix and m_priceMatrix). The following code example may be of help:

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

5.3 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.

5.4 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.

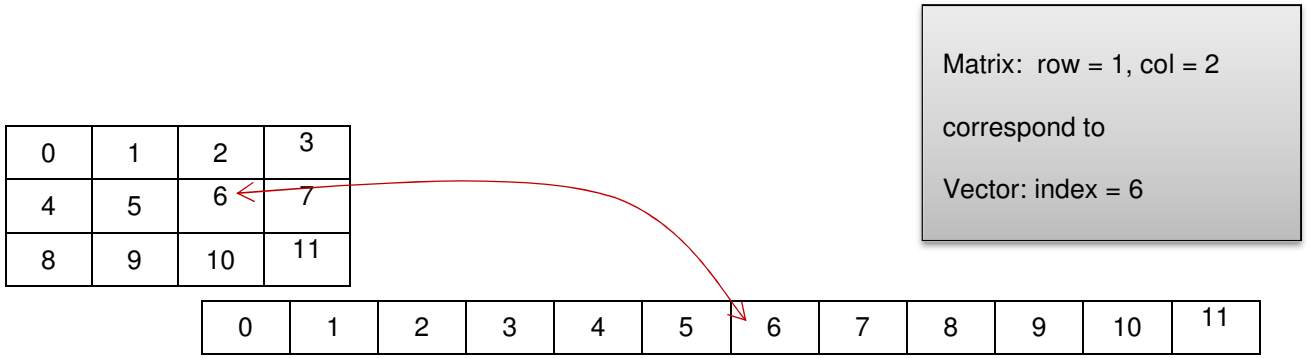
```
public class SeatManager
{
    //fileds
    private readonly int m_totNumOfCols = 0;    //number of rows in the cinema
    private readonly int m_totNumOfRows = 0;    //number of columns per row
    private string[,] m_nameMatrix;             //two-dim array for storing seat names

    /// <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)
```

5.5 When working with two-dimensional arrays, it is quite practical to run a nested loop as below:

```
for (int row = 0; row < m_totNumOfRows; row++)
{
    //You may have code for every row here
    for (int col = 0; col < m_totNumOfCols; col++)
    {
        //here comes code to be executed for
       //every column of the current row.
    }
```

5.5.1 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).



5.5.2 A method that maps an index in a one dimensional array to a row and col in a two dimensional array (the opposite of the above, solution given at the end of this document). The element with index 6 in a one-dimensional array corresponds to row = 1 and col = 2 in a two-dimensional array as illustrated in the above figure.

6. The DisplayOptions enum

6.1 Write a public enum **DisplayOptions** to define a number of choices as in the figure. Save this enum in a separate file DisplayOptions.cs. 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.

This enum can be used in the SeatManager as an option for calculating values or preparing information for each choice (more info below).

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

7. The MainForm Class

7.1 To fill the ComboBox with members with members of the DisplayOption and set a default item, the following code can be used:

```
cmbDisplayOptions.Items.AddRange(Enum.GetNames(typeof(DisplayOptions)));
cmbDisplayOptions.SelectedIndex = (int)DisplayOptions.AllSeats;
```

7.2 It should however be mentioned that there is also another smooth way of filling the ComboBox through binding:

```
cmbDisplayOptions.DataSource = Enum.GetNames(typeof(DisplayOptions));
```

The first item in the enum will be selected automatically in this alternative.

7.3 The GUI should now also manage the input for both rows and columns and **ListBox** should display these too as illustrated in the assignment description.

7.4 Write code in the **InitializeGUI** method to fill the **ComboBox** with items for the **DisplayOption** enum. Two-ways to do that:

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

8. Working with enums in conjunction with ComboBoxes and ListBoxes

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

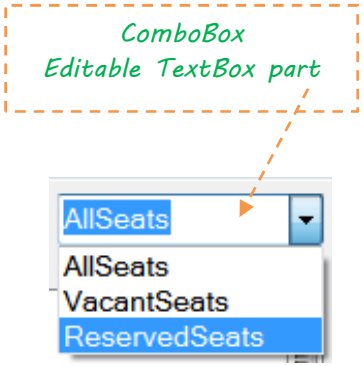
```
(DisplayOptions)cmbDisplayOptions.SelectedIndex
```

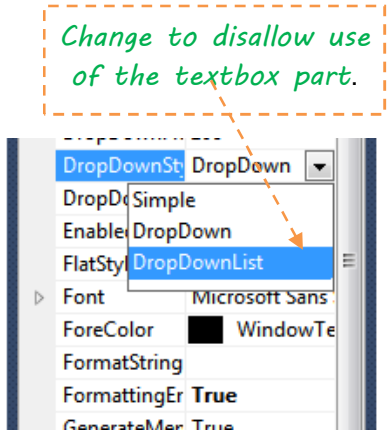
and to convert from enum-element to an int you can do as follows:

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

8.2 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.

8.3 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.





8.4 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)));
```

8.5 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.

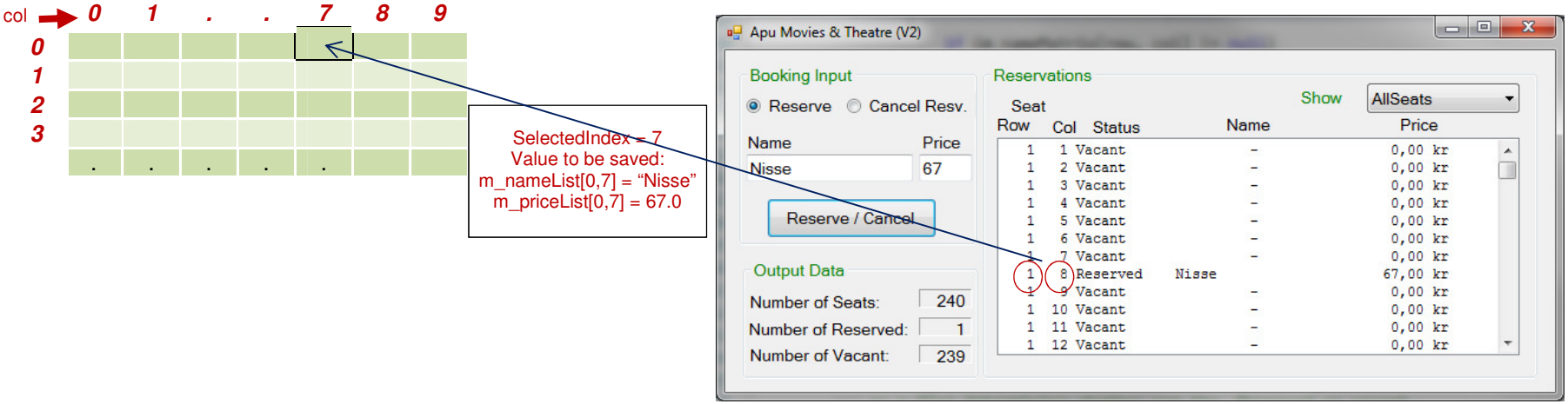
8.6 To select the first entry (option) in the ComboBox as default, set

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

9. Mapping the ListBox items to rows and columns in the SeatManager class

9.1 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

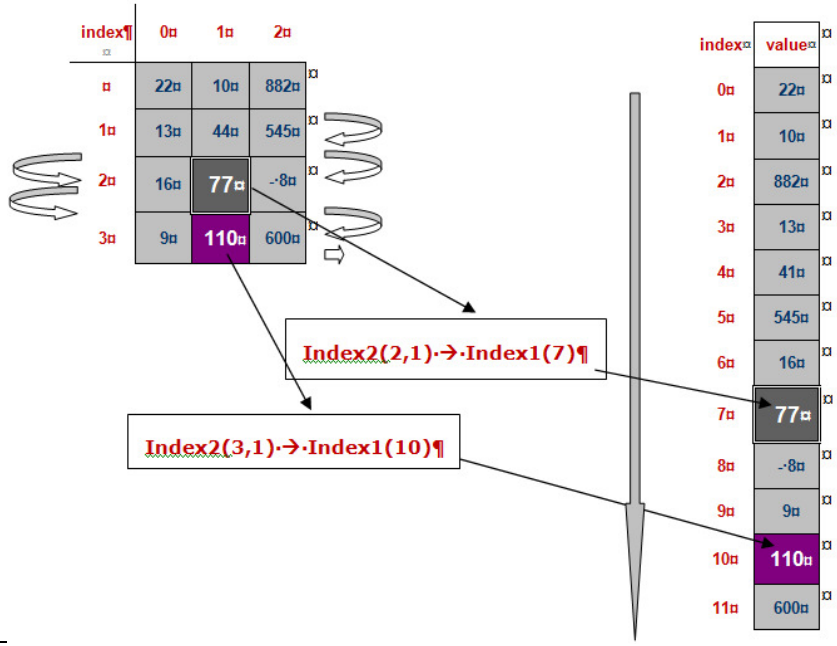
9.2 The hardest task in this version is perhaps to map en entry in the list box (where the items are a one-dimensional array) to an index (row, col) in your matrices. Items on the ListBox are indexed as a one-dimensional array from 0 to number of items -1. This information is available to your code as the value of the ListBox's SelectedIndex. (lstReservations.SelectedIndex = 7 in the run example below).



9.3 The items in the ListBox are all strings that we have formatted in a way so they appear tabulated. So, how would do to fetch the corresponding row and column in your matrices in the SeatManager class? You can of course extract the row and col from the text by writing an algorithm, but that would be both risky and not always efficient.

A good solution might be to write an algorithm that converts the value of the SelectedIndex property to a corresponding pair of row and column numbers as illustrated in the figure above, where the item number 7 (counted from 0 at the top) in the ListBox corresponds to row = 0, and column 7 in the matrices in the SeatManager object..

9.4 To construct an alogrithm for such a mapping, study the the figur here at the right which demonstrates how an index in a one-dimensional array can be mapped to a pair of row and column indices in a two-dimensional array, by rolling out a two-dimensional array to a one-dimensional list! A value saved for example in the position (3, 1) has a position 10 in the one-dimensional array.



9.5 Now, let's try to figure out a little algorithm to change index from a matrix system to one-dimensional array. Assume an item in the ListBox is selected by the user and we must find out in which position (row, col) we should save the name and price for the reservation. Examin the code snippets below:

```
/// <summary>
/// Convert a two-dimensional array (matrix) into a corresponding
/// one-dimensional array. Map a position, row & col,in the
/// m_nameList to an item on a ListBox, having only rows.
/// Calculate the cell indices (row and col) to a corresponding
/// row in the one-dimensional array.
/// </summary>
/// <param name="row">Index of a row in the two-dimensional array.</param>
/// <param name="col">Index of a column in the two-dimensional array</param>
/// <returns>The row index to the corresponding item in the one-
/// dimensional array.</returns>
/// <remarks></remarks>

public int MatrixIndexToVectorIndex(int row, int col)
{
    int index = row * m_totNumOfCols + col;
    return index;
}
```

9.6 To do the opposite in case we would need that, the code below shows the solution:

```
/// <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
}
```

10. Multidimensional arrays vs one-dimensional array of objects

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 data in form of tables representing some type of data, and it is of course more natural to program the tables as they are. Using two or multidimensional arrays has a big disadvantage, and that is readability of the code. Each dimension represents something that is not directly readable from the code. Results[5,6] does not reveal what the rows and columns represent as they don't have a name. It could be Results[month, year], Results[matchnr, and score] or whatever. A good documentation might help a lot.

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 in which we use an array of Seat objects (See also Version 3 later in this documente).

We have now an array of objects indexed from zero to somet thing, 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 one field. This will give you a more maintainable structure.

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.

My experience is that any type of a multidimensional data can be represented to a one-dimensional array, just by rolling them out into a one-dimensional array, just as explained earlier in this document. You can then write a litte function that maps a multi-dimensional array index to a position in a one-dimensional array to access the same value.

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. As a rule of thumb, as soon as you have a two or multidimensional list of data, create a class and declare a field for each dimension. Then use a list of objects instead and that's why the next version, array of object is included in this assignment. Using a list of objects offers not only good readability, it provides a lot of flexibility. You can use different types for the dimensions, you can use variable names for each dimension, and you can easily put more dimensions (fields).

Version 3: 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 encapsulate 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.

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.

11. 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.

12. The Seat Class

Create a new class **Seat** and complete it with fields for the above data and methods necessary to perform its tasks. Use enums to group different type of constant data and make your class prepared for handling more information about a seat.

```
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

}
```

```
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
}
```

13. The SeatManager Class

Create a new class (not much of the previsous 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 using 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.

```
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
    }
```

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.

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;.
```

```
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];

}
```

To do:

Write the following methods:

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

You may need to write other methods too.

Hopefully, this document has given you answers to many of your questions. Use the forums to discuss questions that still remain ambiguous in your thoughts.

Good Luck!

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

Farid Naisan
Course Responsible and Instructor