

10 Theatre Bookings

In this final program, we will bring together many of the programming techniques which you have met earlier in the book, including interactive user interface design, file handling, and the display of graphics images. The project is large and may take several days to complete. However, much of the code is similar to work carried out previously in this book. You may save time by cutting and pasting methods from previous programs you have written, then making any necessary alterations.

Newbridge Theatre

You are asked to produce a seat booking program which could be used by staff working in the box office of the Newbridge Theatre. Customers phone the box office to obtain information about forthcoming events, then may book seats for a particular performance.

Requirements of the system:

- The program should display information and images of events, to help office staff in answering customer enquiries. The dates and times of performances for a particular event should be shown.
- When a performance is selected, a plan of the theatre will be displayed to indicate available seats. The theatre plan is:

				1	2	3	4	5	6	7	8	9	10	11	A	12	13	14				
			1	2	3	4	5	6	7	8	9	10	11	12	B	13	14	15	16			
		1	2	3	4	5	6	7	8	9	10	11	12	13	C	14	15	16	17			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	D	15	16	17	18	19		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	E	16	17	18	19	20		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	F	16	17	18	19	20		
		1	2	3	4	5	6	7	8	9	10	11	12	13	G	15	16	17	18	19		
		1	2	3	4	5	6	7	8	9	10	11	12	13	H	15	16	17	18	19		
		1	2	3	4	5	6	7	8	9	10	11	12	13	J	15	16	17	18	19		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	K	16	17	18	19	20		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	L	16	17	18	19			

- As seats are selected, the total price of tickets will be displayed. The theatre has a policy of charging the same price for all seats at a performance, although seat prices may differ between performances.
- If the customer wishes to proceed with a booking, their name, address, e-mail and telephone details will be required. For existing customers, these details can be selected from a database. For new customers, the details must be entered into the system.

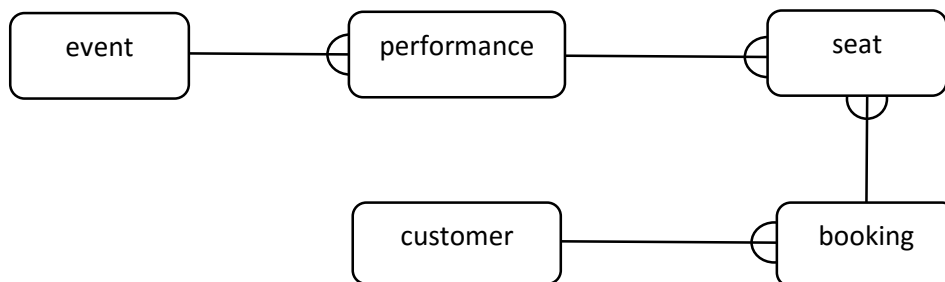
- The customer will confirm their booking by providing credit card details. (Note: obtaining payment from the customer's bank and delivering tickets to the customer are outside the scope of the system which you are asked to produce.)
- Theatre staff should be able to review the bookings received and the total value of ticket sales for any performance.
- Staff should be able to add new events and performances to the system.

Design

As for the **College Courses** program in chapter 8, we will separate the code into two categories:

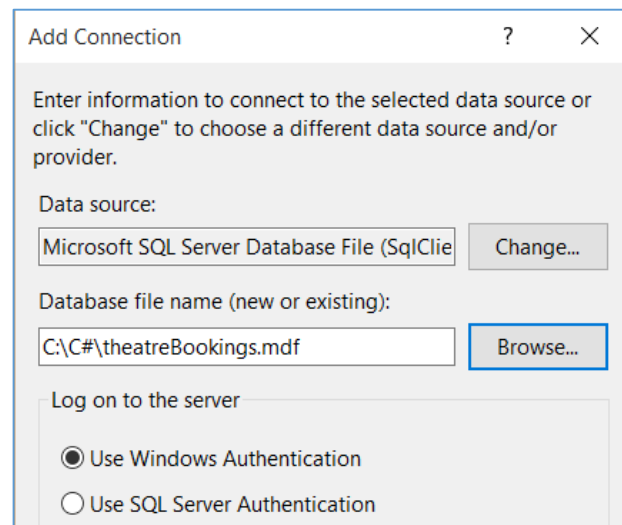
- A set of **Windows Forms** will be used for on-screen *input* by the user and for the *output* of data using text or graphics as required.
- A set of **Object Classes** will be used to handle all *database operations*, including the loading, saving and updating of records.

A good starting point for a complex project is to identify classes of objects for the data model. We will use the following structure:

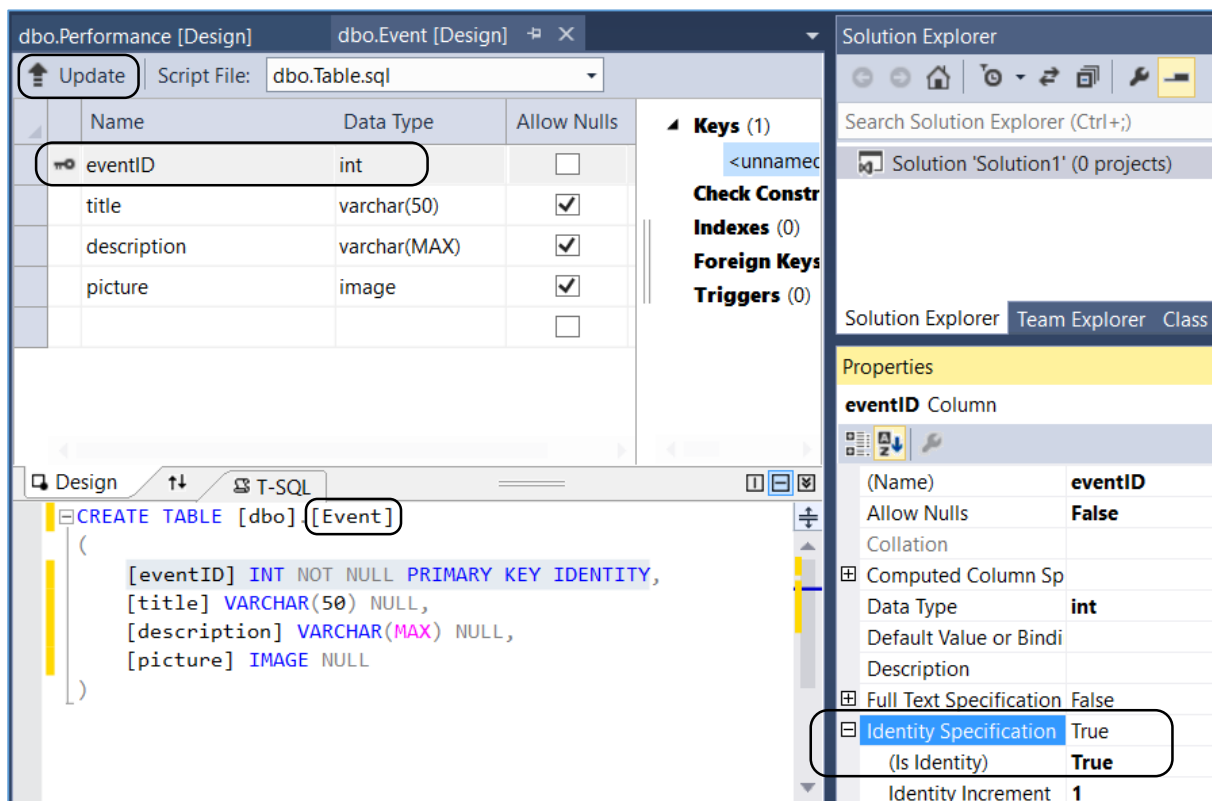


- An **event** may have a number of **performances** on different days or at different times.
- Each **performance** will have a complete set of **seats** available in the theatre.
- A **customer** may make one or more **bookings**.
- Each **booking** will be for a particular **performance**, and may be for a number of **seats**.

Begin the project by creating a '*theatreBookings*' database



Set up a table for storing data for each of the object classes. Begin with an *Event* table:



Make the *eventID* field an auto-number by selecting '*Identity Specification*' and setting the '*(Is Identity)*' property to '*True*'.

When the table design is completed, click the '*Update*' button to add the table to the database.

Add a **Performance** table. Set the **performanceID** field to be an auto-number.

The screenshot shows the SQL Server Enterprise Designer interface. The top pane displays the table design for **dbo.Performance**. The columns are:

Name	Data Type	Allow Nulls
performanceID	int	<input type="checkbox"/>
eventID	int	<input checked="" type="checkbox"/>
performanceDate	datetime	<input checked="" type="checkbox"/>
time	varchar(50)	<input checked="" type="checkbox"/>
seatPrice	decimal(18,2)	<input checked="" type="checkbox"/>

The bottom pane shows the T-SQL script for creating the table:

```
CREATE TABLE [dbo].[Performance]
(
    [performanceID] INT NOT NULL PRIMARY KEY IDENTITY,
    [eventID] INT NULL,
    [performanceDate] DATETIME NULL,
    [time] VARCHAR(50) NULL,
    [seatPrice] DECIMAL(18, 2) NULL
)
```

The right pane shows the Properties window for the **performanceID** column:

Property	Value
(Name)	performanceID
Allow Nulls	False
Collation	
Computed Column Specification	
Data Type	int
Default Value or Binding	
Description	
Full Text Specification	False
Identity Specification (Is Identity)	True
Identity Increment	1

Add a **Seat** table. A primary key does not need to be set, so delete the key from the first line of the table by right-clicking on the 'key' icon:

The screenshot shows the SQL Server Enterprise Designer interface for the **dbo.Seat** table. The top pane displays the table design:

Name	Data Type	Allow Nulls	Default
seatRow	char(1)	<input type="checkbox"/>	
seatNumber	int	<input checked="" type="checkbox"/>	
performanceID	int	<input checked="" type="checkbox"/>	
available	int	<input checked="" type="checkbox"/>	
bookingID	int	<input checked="" type="checkbox"/>	

The bottom pane shows the T-SQL script for creating the table:

```
CREATE TABLE [dbo].[Seat]
(
    [seatRow] CHAR NOT NULL,
    [seatNumber] INT NULL,
    [performanceID] INT NULL,
    [available] INT NULL,
    [bookingID] INT NULL
)
```

Add a **Customer** table, making the **customerID** field an auto-number:

The screenshot shows the SQL Server Enterprise Designer interface. The main window displays the design of the **Customer** table. The columns are listed in a table with headers: Name, Data Type, and Allow Nulls. The **customerID** column is highlighted, and its properties are shown in the Properties window on the right. The T-SQL view at the bottom shows the **CREATE TABLE** statement for **Customer**.

Name	Data Type	Allow Nulls
customerID	int	<input type="checkbox"/>
forename	varchar(50)	<input checked="" type="checkbox"/>
surname	varchar(50)	<input checked="" type="checkbox"/>
title	varchar(20)	<input checked="" type="checkbox"/>
address1	varchar(50)	<input checked="" type="checkbox"/>
address2	varchar(50)	<input checked="" type="checkbox"/>
town	varchar(50)	<input checked="" type="checkbox"/>
postcode	varchar(12)	<input checked="" type="checkbox"/>
email	varchar(50)	<input checked="" type="checkbox"/>
phone	varchar(50)	<input checked="" type="checkbox"/>

```

CREATE TABLE [dbo].[Customer]
(
    [customerID] INT NOT NULL PRIMARY KEY IDENTITY,
    [forename] VARCHAR(50) NULL,
    [surname] VARCHAR(50) NULL,
)
  
```

Properties window for **customerID** Column:

Property	Value
(Name)	customerID
Allow Nulls	False
Collation	
Computed Column Sp	
Data Type	int
Default Value or Bind	
Description	
Full Text Specification	False
Identity Specification	True
(Is Identity)	True
Identity Increment	1

We finally require a **Booking** table. Make the **bookingID** field an auto-number:

The screenshot shows the SQL Server Enterprise Designer interface. The main window displays the design of the **Booking** table. The columns are listed in a table with headers: Name, Data Type, and Allow Nulls. The **bookingID** column is highlighted, and its properties are shown in the Properties window on the right. The T-SQL view at the bottom shows the **CREATE TABLE** statement for **Booking**.

Name	Data Type	Allow Nulls
bookingID	int	<input type="checkbox"/>
customerID	int	<input checked="" type="checkbox"/>
performanceID	int	<input checked="" type="checkbox"/>
totalCost	numeric(18,2)	<input checked="" type="checkbox"/>
creditCardNo	varchar(50)	<input checked="" type="checkbox"/>

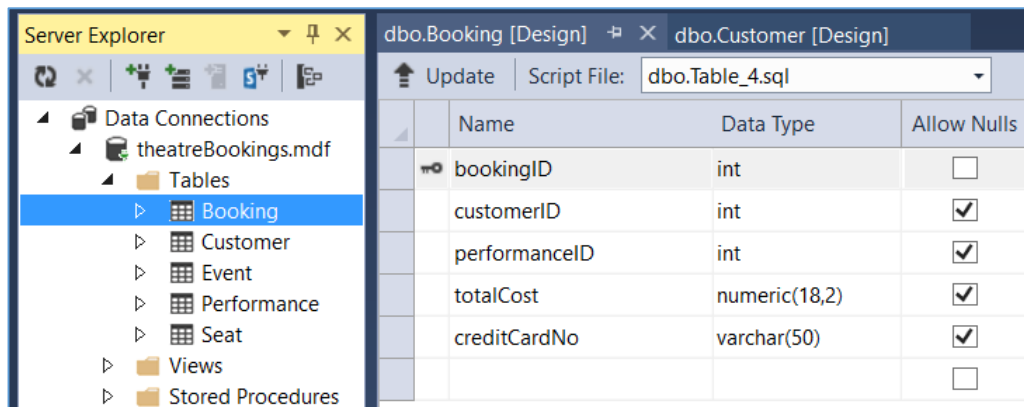
```

CREATE TABLE [dbo].[Booking]
(
    [bookingID] INT NOT NULL PRIMARY KEY IDENTITY,
    [customerID] INT NULL,
    [performanceID] INT NULL,
    [totalCost] NUMERIC(18, 2) NULL,
    [creditCardNo] VARCHAR(50) NULL
)
  
```

Properties window for **bookingID** Column:

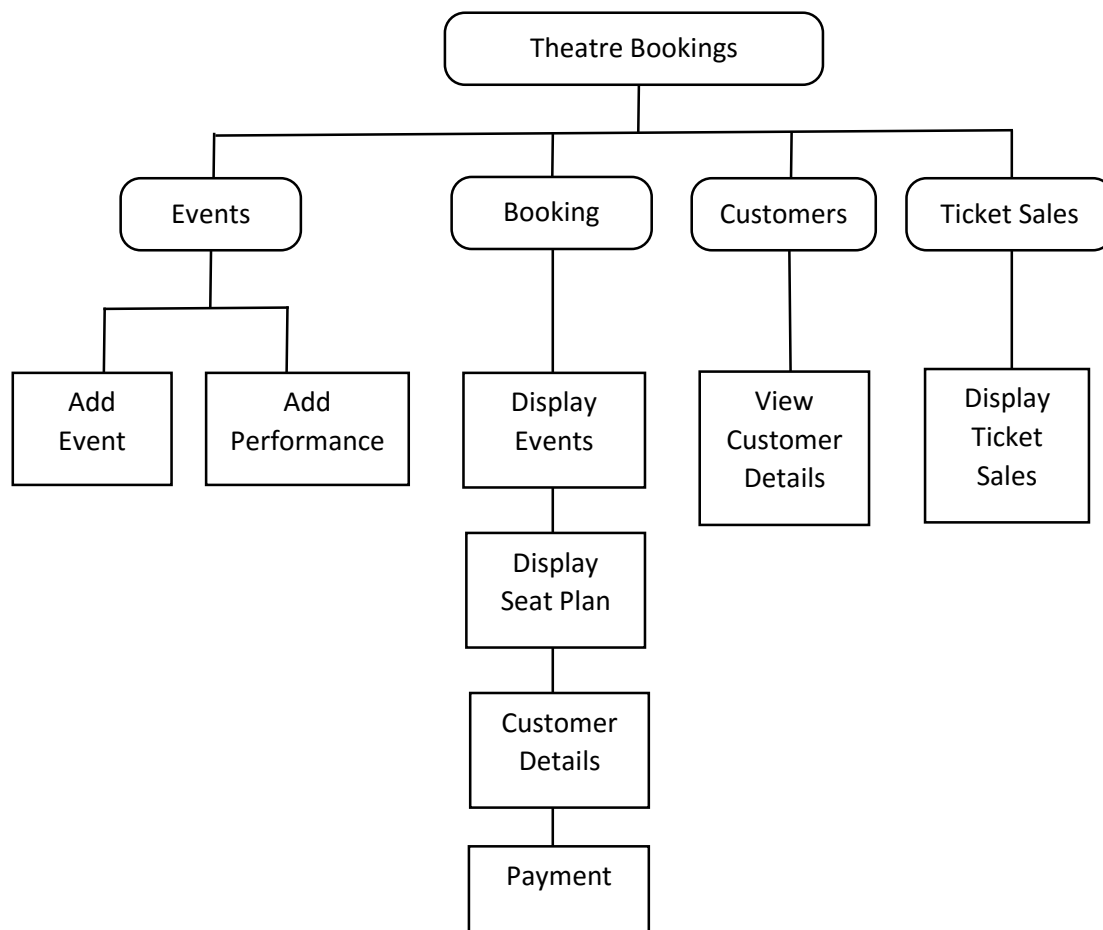
Property	Value
(Name)	bookingID
Allow Nulls	False
Collation	
Computed Column Sp	
Data Type	int
Default Value or Bind	
Description	
Full Text Specification	False
Identity Specification	True
(Is Identity)	True
Identity Increment	1

Click the 'Refresh' icon in the Server Explorer window, then check that all the tables have been created correctly:

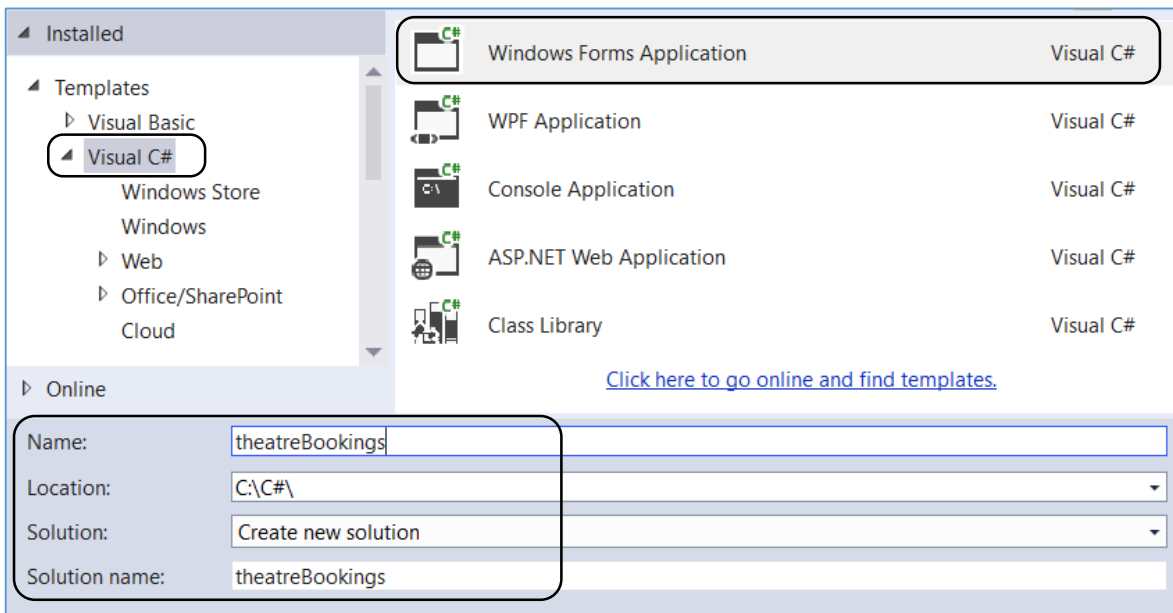


Right-click on **theatreBookings.mdf** and delete the connection to the database.

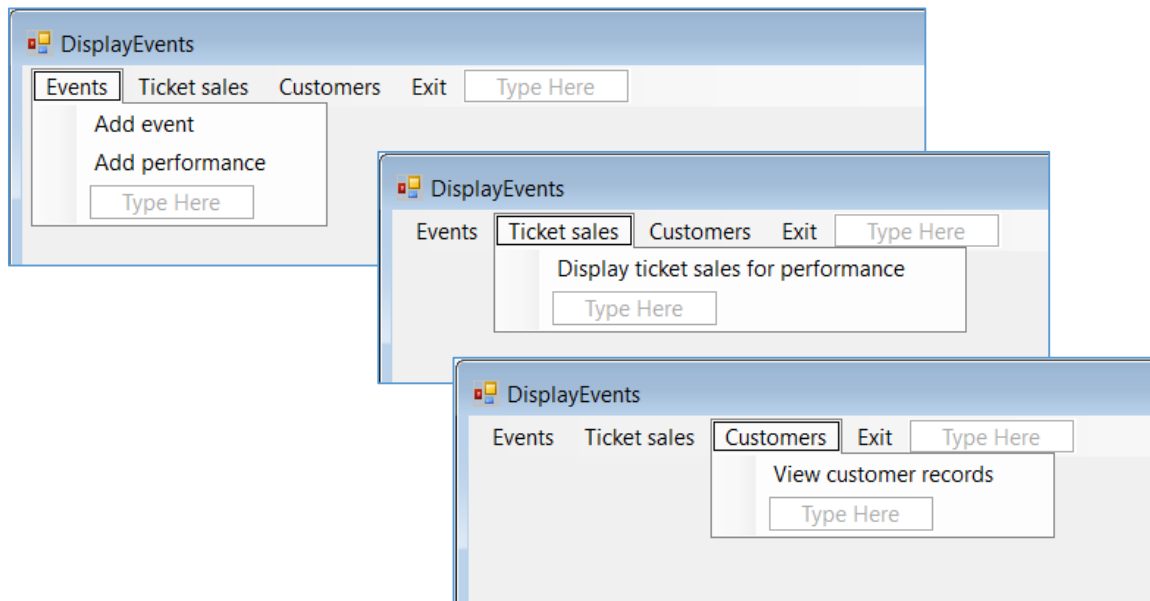
We can now plan how the Windows forms of the project will be related. From the specification it seems that we will need four program sections: to handle the input of new **event and performance details**, to **process a booking** by a customer, to **display customer records** for the theatre staff, and to display the **seat bookings and ticket sales** for each performance.



Start a new Visual C# project. Select **Windows Forms Application** and give the name *theatreBookings*.



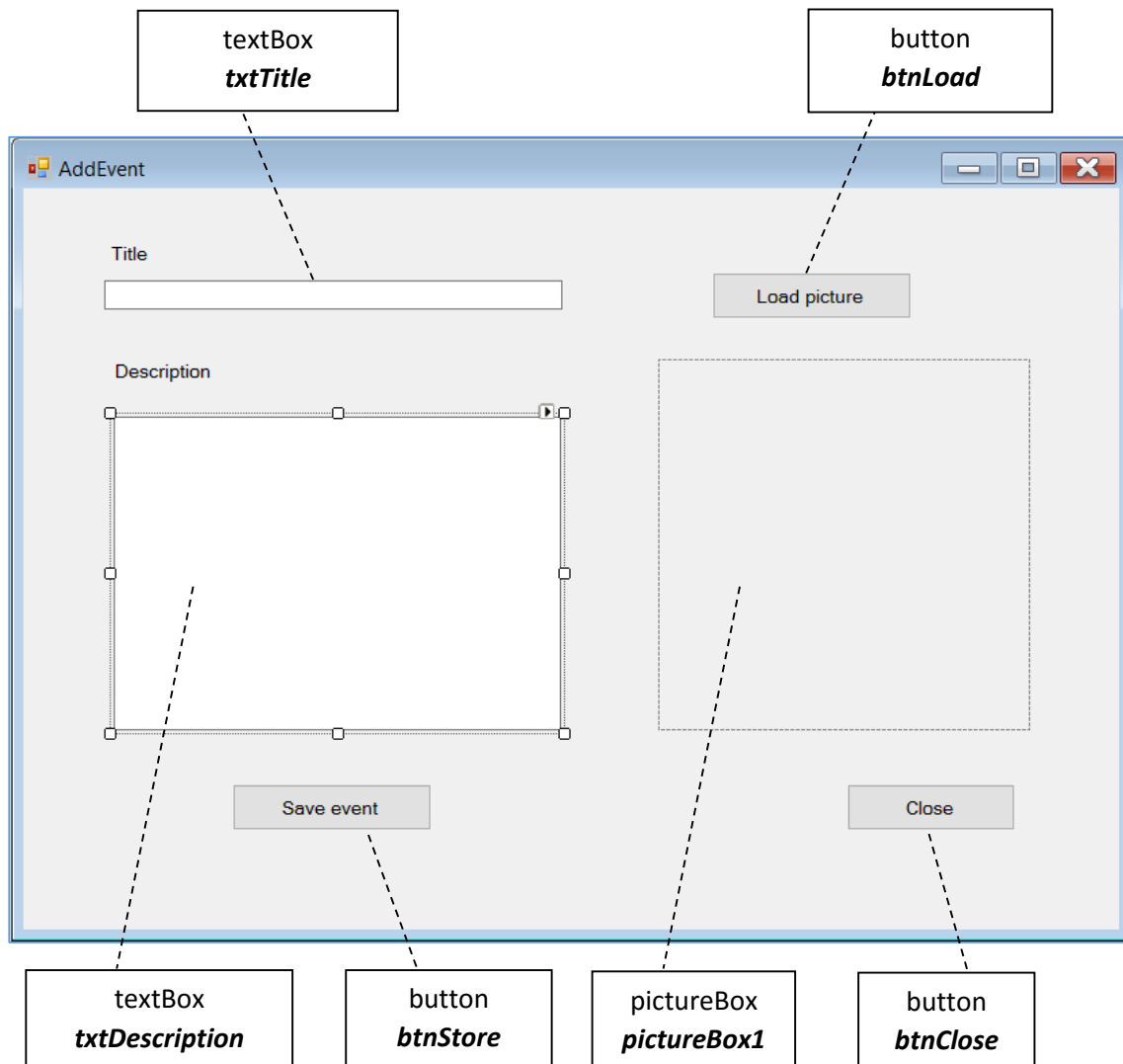
Rename Form1 as *DisplayEvents*. Add a *menuStrip* component, and configure the menu options as shown:



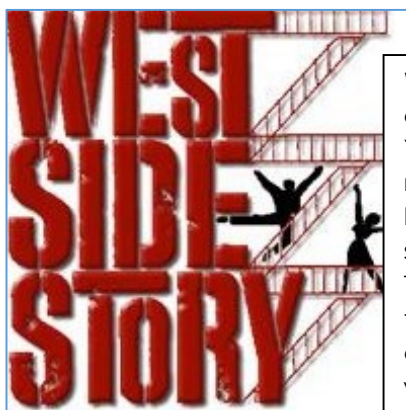
Add a Windows Form and give this the name '*AddEvent*'. Link the *AddEvent* form to the menu system by adding code to the '*Add event*' menu option:

```
private void addEventToolStripMenuItem_Click(object sender, EventArgs e)
{
    AddEvent frmAddEvent = new AddEvent();
    frmAddEvent.ShowDialog();
}
```

Add components to the **AddEvent** form. For the **txtDescription** text box, set the **Multiline** property to **True**.



For each theatre event, it would be good to provide a picture image and a written description of the event. Go to the Internet and find suitable images and text for some events that the theatre might host:



West Side Story is set in the East 40s and West 50s of the Upper West Side neighborhood in New York City in the mid-1950s, an ethnic, blue-collar neighborhood. The musical explores the rivalry between the Jets and the Sharks, two teenage street gangs of different ethnic backgrounds. The members of the Sharks from Puerto Rico are taunted by the Jets, a Polish-American working-class group. Tony, one of the Jets, falls in love with Maria, the sister of Bernardo, the leader of the Sharks.

Add code for the **Load** and **Close** buttons on the **AddEvent** form. A variable called `imagename` is also required:

```
public partial class AddEvent : Form
{
    string imagename;

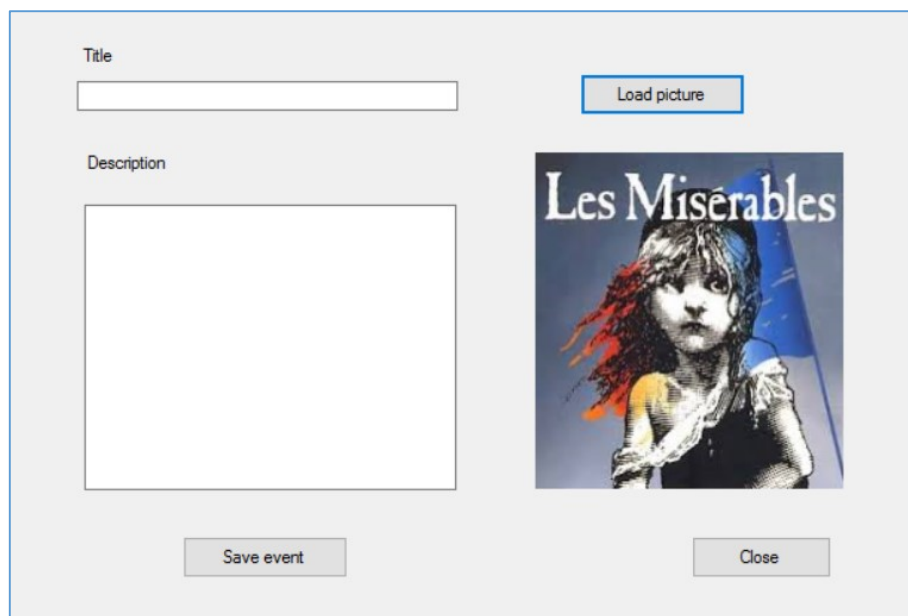
    public AddEvent()
    {
        InitializeComponent();
    }

    private void btnClose_Click(object sender, EventArgs e)
    {
        this.Close();
    }

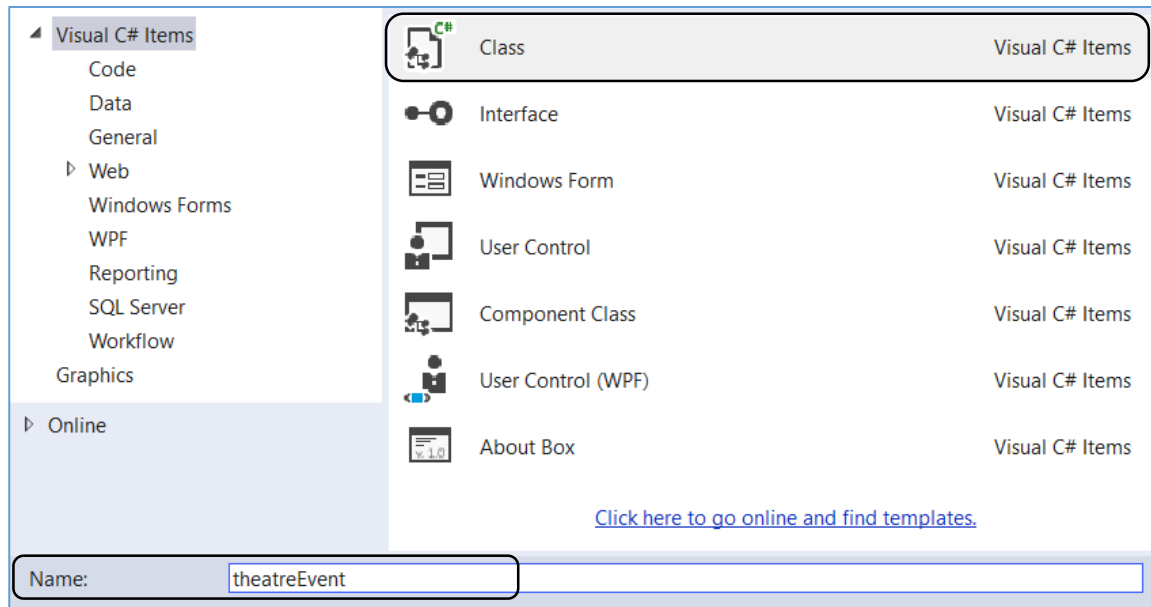
    private void btnLoad_Click(object sender, EventArgs e)
    {
        try
        {
            OpenFileDialog openFileDialog = new OpenFileDialog();
            openFileDialog.Filter = "Image File (*.jpg;*.bmp;*.gif)|*.jpg;*.bmp;*.gif";

            if (openFileDialog.ShowDialog() == DialogResult.OK)
            {
                imagename = openFileDialog.FileName;
                Bitmap newimg = new Bitmap(imagename);
                pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
                pictureBox1.Image = (Image)newimg;
            }
            openFileDialog = null;
        }
        catch
        {
            MessageBox.Show("Error");
        }
    }
}
```

Run the program. Select the **'Add event'** menu option to open the **AddEvent** form. Click the **'Load picture'** button, and check that a picture image can be selected and displayed.



Create a class file called '***theatreEvent***'. (Note: It is not possible to create a class called '***event***', as the word 'event' has a special meaning in the C# language.)



Open the ***theatreEvent*** class file and add the properties for a ***theatreEvent*** object. Since we are using picture images, it is necessary to include a '***using Drawing***' directive:

```
using System.Linq;
using System.Text;

using System.Drawing;

namespace theatreBookings
{
    class theatreEvent
    {
        private int eventID;
        private string title;
        private string description;
        private Image imageData;
    }
}
```

We will now add the series of methods required to move data values into or out of each property field of the ***theatreEvent*** objects.

```
private string description;
private Image imageData;

public void setEventID(int e)
{
    eventID = e;
}

public int getEventID()
{
    return eventID;
}
```

```

    public void setTitle(string t)
    {
        title = t;
    }

    public string getTitle()
    {
        return title;
    }

    public void setDescription(string d)
    {
        description = d;
    }

    public string getDescription()
    {
        return description;
    }

    public void setImage(Image im)
    {
        imageData = im;
    }

    public Image getImage()
    {
        return imageData;
    }
}

```

We need to create a method to save theatreEvent records into the database. Before doing that, a few more **'using'** directives will be needed, and the database location must be specified. You should also set up a variable to keep a **count** of the number of **theatreEvent objects** in the system, and an **array** to link to these **theatreEvent objects**.

```

using System.Text;
using System.Drawing;

using System.IO;
using System.Data.SqlClient;
using System.Data;

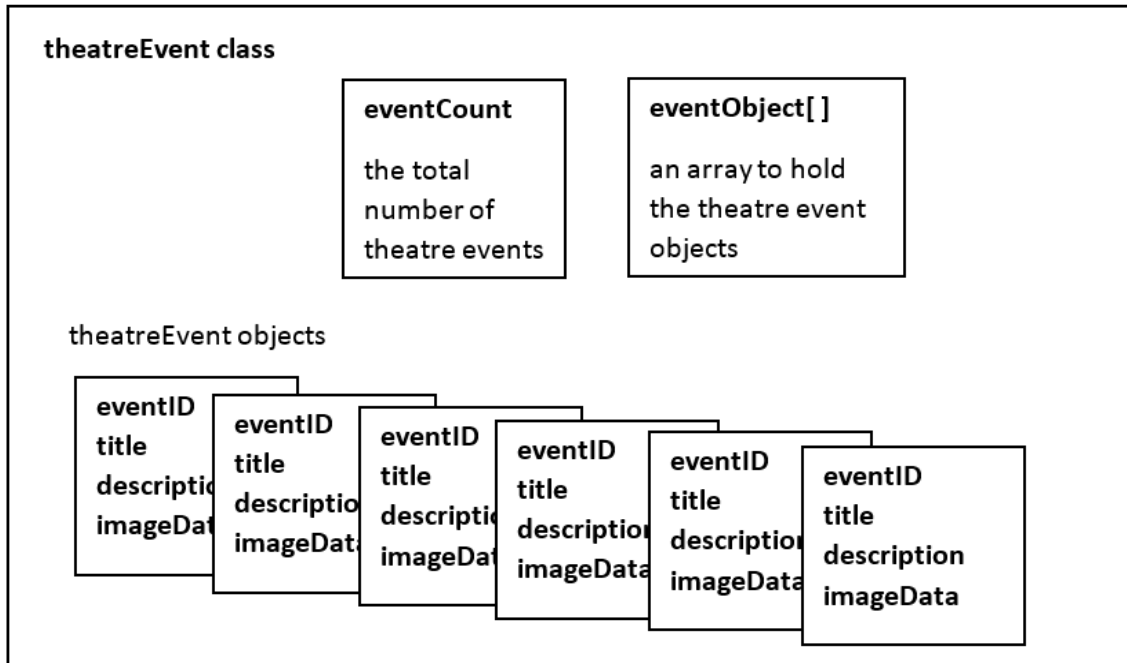
namespace theatreBookings
{
    class theatreEvent
    {
        private static string databaseLocation = "C:\\C#\\theatreBookings.mdf";
        public static int eventCount;
        public static theatreEvent[] eventObject = new theatreEvent[12];

        private int eventID;
        private string title;
        private string description;
        private Image imageData;
    }
}

```

Notice that the variables '**eventCount**' and '**eventObject**' have been marked as '**static**'. This means that they occur only once and are used by the whole class.

By contrast, the properties **eventID**, **title**, **description** and **imageData** are '**dynamic**': a set of these variables is created for each new object added whilst the program is running.



We will now add an **AddEvent()** method to the **theatreEvent** class. This will convert the picture image into an array of binary data, then save it into a database record, along with the event title and description. The same technique was used to store the **Fast Food** images in chapter 9. Insert the new method below the list of theatreEvent properties.

```
public static void AddEvent(string im, string t, string d)
{
    FileStream fs;
    fs = new FileStream(im, FileMode.Open, FileAccess.Read);
    byte[] picbyte = new byte[fs.Length];
    fs.Read(picbyte, 0, System.Convert.ToInt32(fs.Length));
    fs.Close();

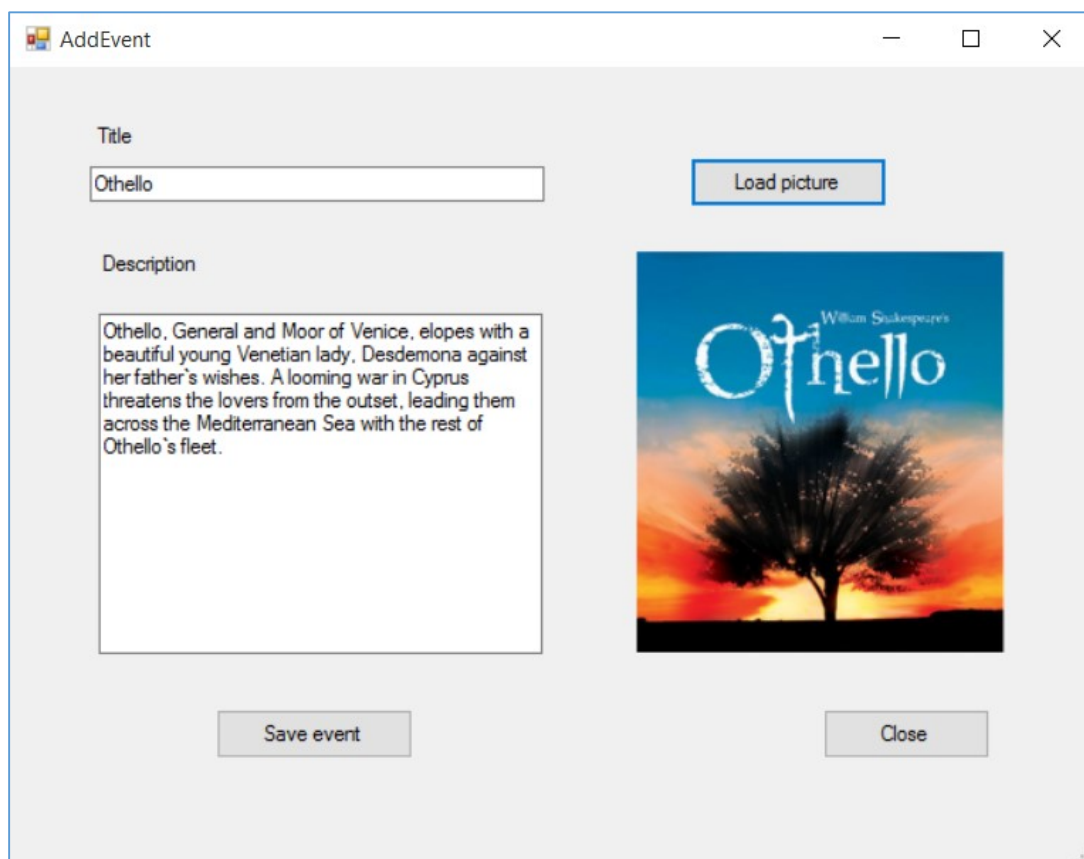
    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;
        AttachDbFilename="+ databaseLocation + "Integrated Security=True;
        Connect Timeout=30; User Instance=True");
    con.Open();
    string query = "INSERT INTO Event(title,description,picture) "+
        "VALUES(' + t + "',' + d + "',' + " @pic)";
    SqlParameter picparameter = new SqlParameter();
    picparameter.SqlDbType = SqlDbType.Image;
    picparameter.ParameterName = "pic";
    picparameter.Value = picbyte;
    SqlCommand cmd = new SqlCommand(query, con);
    cmd.Parameters.Add(picparameter);
    cmd.ExecuteNonQuery();
    con.Close();
}
```

Return to the **AddEvent** form and double click the 'Save event' button. Include a line of code in the button click method to call an **AddRecord()** method, then add this method:

```
private void btnStore_Click(object sender, EventArgs e)
{
    addRecord();
}

private void addRecord()
{
    try
    {
        if (imagename != "")
        {
            theatreEvent.AddEvent(imagename, txtTitle.Text, txtDescription.Text);
            MessageBox.Show("Event Added");
        }
    }
    catch
    {
        MessageBox.Show("File error");
    }
}
```

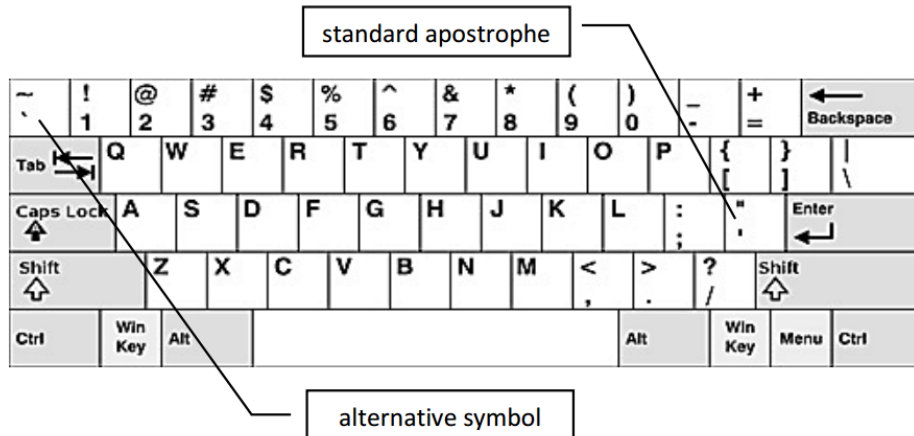
Notice how this method calls **AddEvent()** in the **theatreEvent** class to save the event. We simply pass the necessary data to the **AddEvent()** method as a series of parameters.



A problem might occur when entering titles or descriptions of events if **apostrophe** characters (') are present in the text, for example:

"The songs include 'Sherry', 'Walk Like A Man' and 'Big Girls Don't Cry'."

The apostrophe is used as a special control character by the C# language, and can cause an error when data is being uploaded to the database. Fortunately there is a simple solution. The computer has an alternative symbol which looks similar to an apostrophe, but is not recognised as a C# control character. This is located in the upper left hand corner of the keyboard.



Return to the **addRecord()** method in the **AddEvent** form, and insert lines of code to make the replacements from the standard apostrophe to the alternative symbol:

standard apostrophe alternative symbol

Replace("'", "`")

```
private void addRecord()
{
    try
    {
        txtTitle.Text = txtTitle.Text.Replace("'", "`");
        txtDescription.Text = txtDescription.Text.Replace("'", "`");

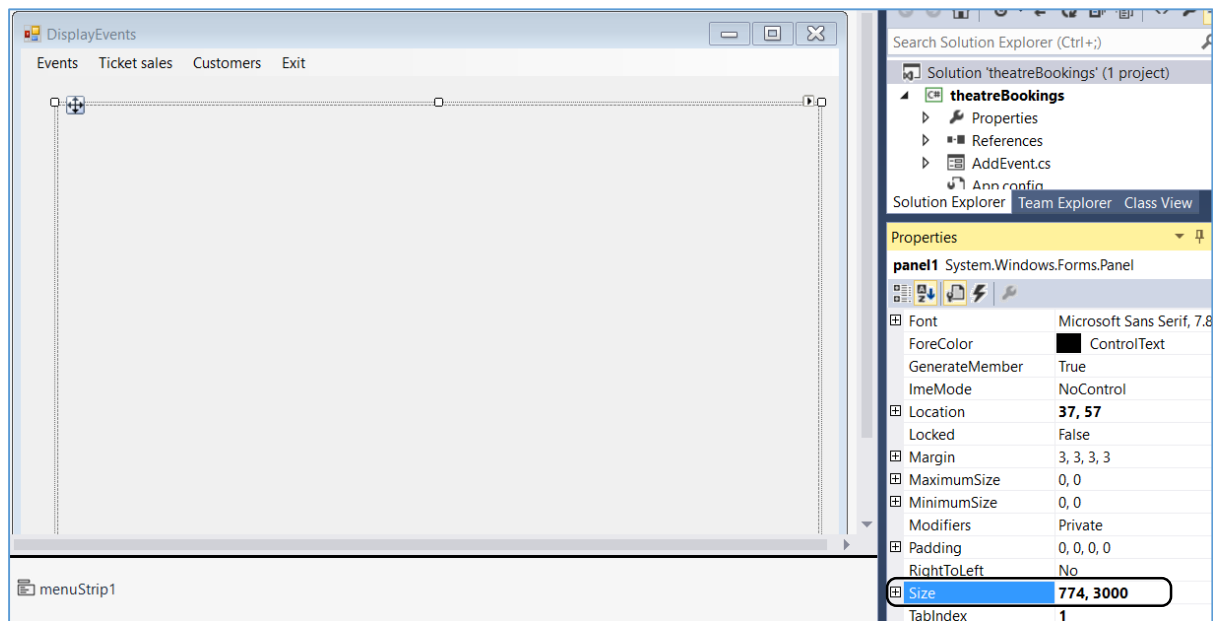
        if (imagenam != "")
        {
            theatreEvent.AddEvent(imagenam, txtTitle.Text, txtDescription.Text);
            MessageBox.Show("Event Added");
        }
    }
    catch
    {
        MessageBox.Show("File error");
    }
}
```

Add a series of event records and check that these are being stored correctly in the database **Event** table:

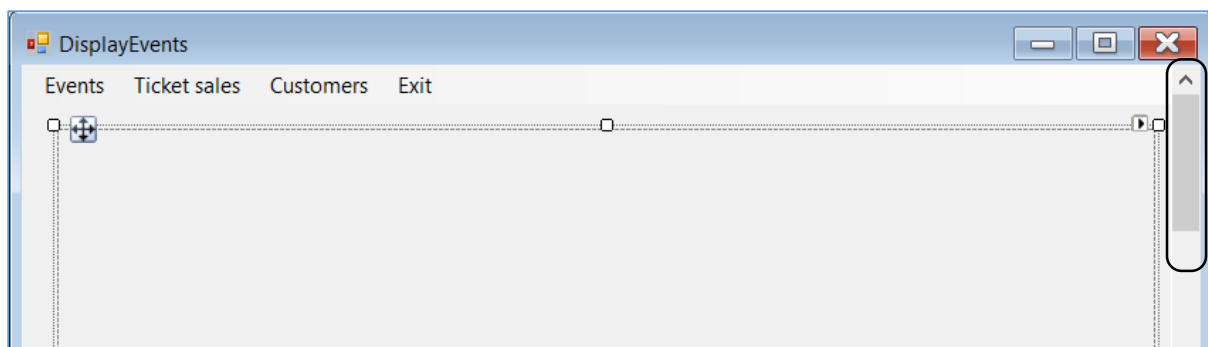
eventID	title	description	picture
1	Les Miserables	Concerns love and bravery in 19th century France during the revolutionary...	<Binary data>
2	The Lion King	When the young lion prince Simba is born his evil uncle Scar is pushed ba...	<Binary data>
3	Jersey Boys	The story of Frankie Valli and The Four Seasons – Frankie Valli, Bob Gaudi...	<Binary data>
4	We Will Rock You	The musical does not chronicle the story of the band, Queen, rather it inc...	<Binary data>
5	Thriller	Pays tribute to the musical hits and career of Michael Jackson and The Jac...	<Binary data>
6	West Side Story	West Side Story is set in the East 40s and West 50s of the Upper West Side...	<Binary data>
7	Othello	Othello, General and Moor of Venice, elopes with a beautiful young Venet...	<Binary data>
NULL	NULL	NULL	NULL

We can now return to the **DisplayEvents** form and work on the task of displaying the event text and pictures.

Add a panel to the **DisplayEvents** form and name this '**pnlEvents**'. Set the height of the panel to 3,000 pixels.



Click once on the **DisplayEvents** form beyond the edge of the panel to select it, then set the '**AutoScroll**' property to '**True**'. Run the program to check that a scrolling window has been produced on the **DisplayEvents** form.



We must now return to the **theatreEvent** class file to add a method to load the event records from the database. Insert the **loadEvents()** method below the list of properties:

```
private int eventID;
private string title;
private string description;
private Image imageData;

public static void loadEvents()
{
    SqlDataAdapter dAdapt;
    DataSet dSet;

    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;
        AttachDbFilename="+ databaseLocation+ "Integrated Security=True;
        Connect Timeout=30; User Instance=True");
    con.Open();
    dAdapt = new SqlDataAdapter();
    dAdapt.SelectCommand = new SqlCommand("SELECT * FROM Event", con);
    dSet = new DataSet("dSet");
    dAdapt.Fill(dSet);
    con.Close();
    DataTable dTable;
    dTable = dSet.Tables[0];

    DataTable dataTable = dSet.Tables[0];
    eventCount = dataTable.Rows.Count;

    for (int i = 0; i < eventCount; i++)
    {
        eventObject[i] = new theatreEvent();
        DataRow dataRow = dataTable.Rows[i];
        string finalString = "pic" + Convert.ToString(i);
        FileStream FS1 = new FileStream(finalString + ".jpg", FileMode.Create);
        byte[] blob = (byte[])dataRow[3];
        FS1.Write(blob, 0, blob.Length);
        FS1.Close();
        FS1 = null;

        eventObject[i].setImage(Image.FromFile(finalString + ".jpg"));
        eventObject[i].setEventID((int)dataRow[0]);
        eventObject[i].setTitle(Convert.ToString(dataRow[1]));
        eventObject[i].setDescription(Convert.ToString(dataRow[2]));
    }
}
```

This method accesses the database, then uses a **loop** to create an **eventObject** from each event **record** in the database. The picture data is converted into **JPG image** format so that it can be easily displayed.

Go to the **DisplayEvents** form and add code to the **DisplayEvents()** method. This calls the **loadEvents()** method in the **theatreEvent** class, to load the data from the database and set up an **eventObject** for each theatre event.

```
public DisplayEvents()
{
    InitializeComponent();

    theatreEvent.eventCount = 0;
    try
    {
        theatreEvent.loadEvents();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Run the program and check that no error message is displayed. If all is well, we can now display the events on the panel. Write a **displayPictures()** method, and call this from the **DisplayEvents()** method.

```
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}

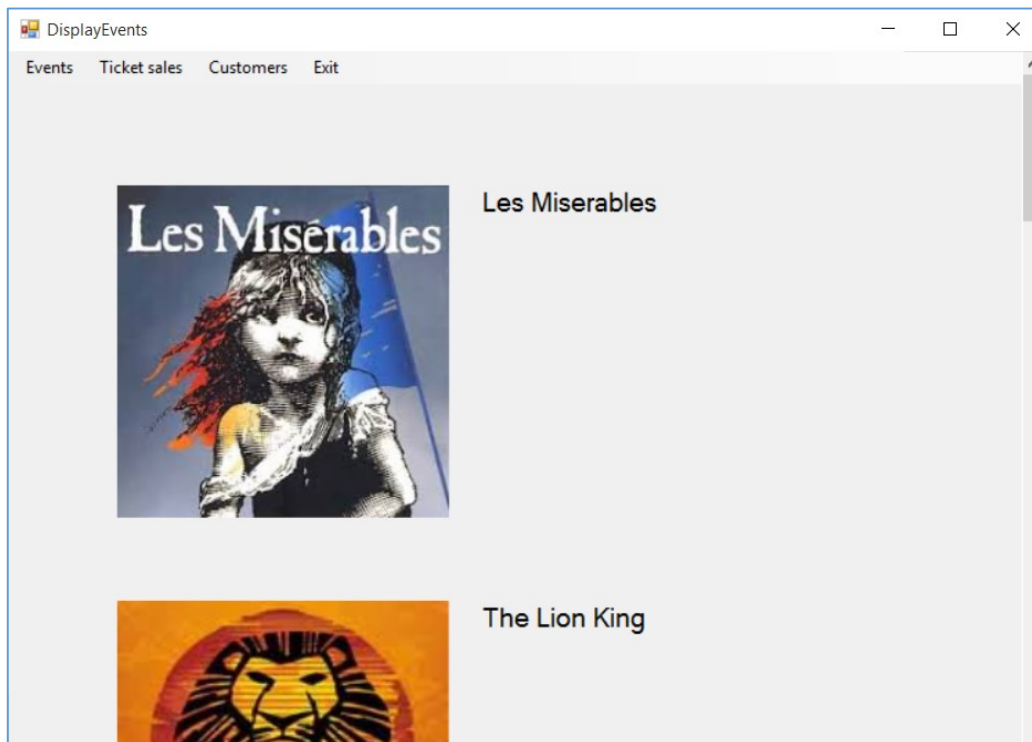
displayPictures();

private void displayPictures()
{
    PictureBox[] pictureBox = new PictureBox[8];
    Label[] label = new Label[8];
    TextBox[] textBox = new TextBox[8];
    Button[] button = new Button[8];

    for (int i = 0; i < theatreEvent.eventCount; i++)
    {
        pictureBox[i] = new PictureBox();
        pictureBox[i].Image = theatreEvent.eventObject[i].getImage();
        pictureBox[i].Size = new System.Drawing.Size(240, 240);
        pictureBox[i].Location = new System.Drawing.Point(60, 60 + 300 * i);
        pictureBox[i].SizeMode = PictureBoxSizeMode.StretchImage;
        pnlEvents.Controls.Add(pictureBox[i]);
        pictureBox[i].Refresh();

        label[i] = new Label();
        label[i].Size = new System.Drawing.Size(200, 30);
        label[i].Text = theatreEvent.eventObject[i].getTitle();
        label[i].Font = new System.Drawing.Font("Microsoft Sans Serif", 14F,
            System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point,
            ((byte)0));
        label[i].Location = new System.Drawing.Point(320, 60 + 300 * i);
        pnlEvents.Controls.Add(label[i]);
    }
}
```

Run the program. The titles and images for your theatre events should be displayed in the scrolling window.



Return to the **displayPictures()** method and add two more sections of code. These will produce a textBox to display the description of the event, and a button which can be clicked to go to the bookings screen.

```
label[i].Location = new System.Drawing.Point(320, 60 + 300 * i);
pnlEvents.Controls.Add(label[i]);

textBox[i] = new TextBox();
textBox[i].TabStop = false;
textBox[i].BorderStyle = System.Windows.Forms.BorderStyle.None;
textBox[i].Location = new System.Drawing.Point(320, 100 + 300 * i);
textBox[i].Multiline = true;
textBox[i].Size = new System.Drawing.Size(500, 200);
textBox[i].Text = theatreEvent.eventObject[i].getDescription();
textBox[i].ReadOnly = true;
textBox[i].Font = new System.Drawing.Font("Microsoft Sans Serif", 10F,
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point,
    ((byte)0));
pnlEvents.Controls.Add(textBox[i]);

button[i] = new Button();
button[i].Location = new System.Drawing.Point(700, 60 + 300 * i);
button[i].Size = new System.Drawing.Size(112, 28);
button[i].Text = "Book seats";
String buttonName = "btn" + i;
button[i].Name = buttonName;
button[i].Click += new EventHandler(loadPlan);
pnlEvents.Controls.Add(button[i]);

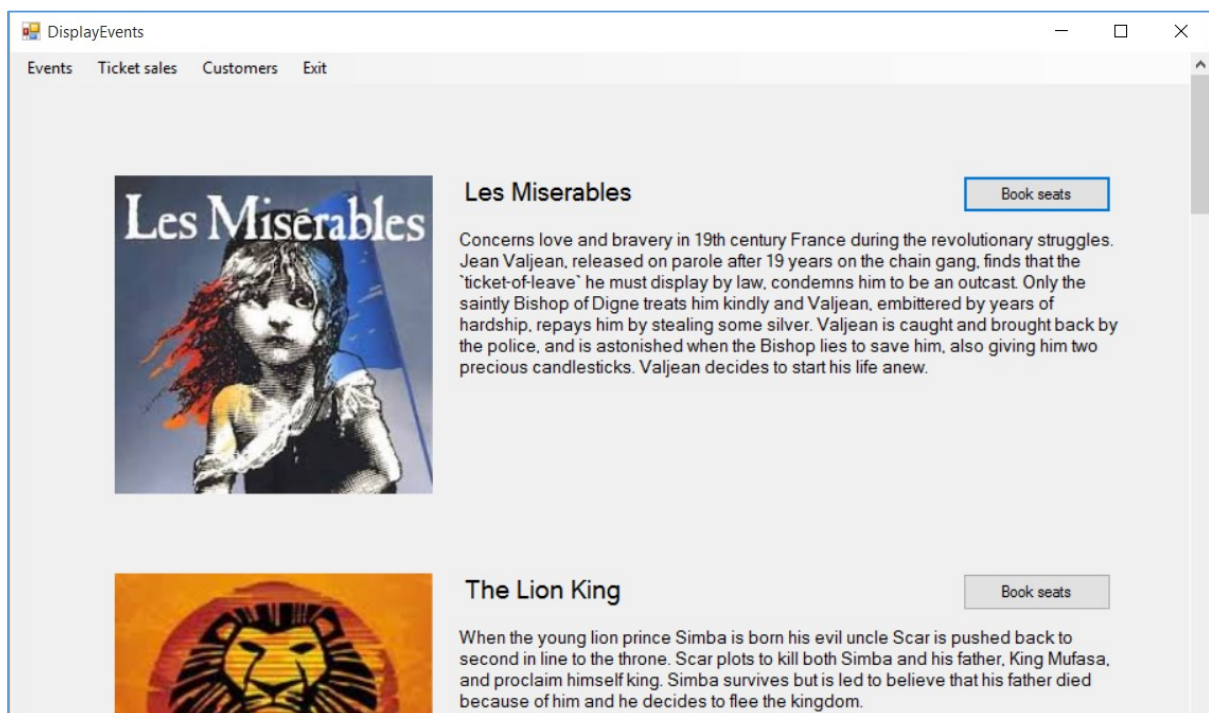
}
}
```

When the '**Book seats**' button is clicked, this will call a **loadPlan()** method, to display a seating plan of the theatre. For now, just create an empty **loadPlan()** method immediately after the **displayPictures()** method. We will come back to complete this later.

```
private void loadPlan(object sender, EventArgs e)
{
}

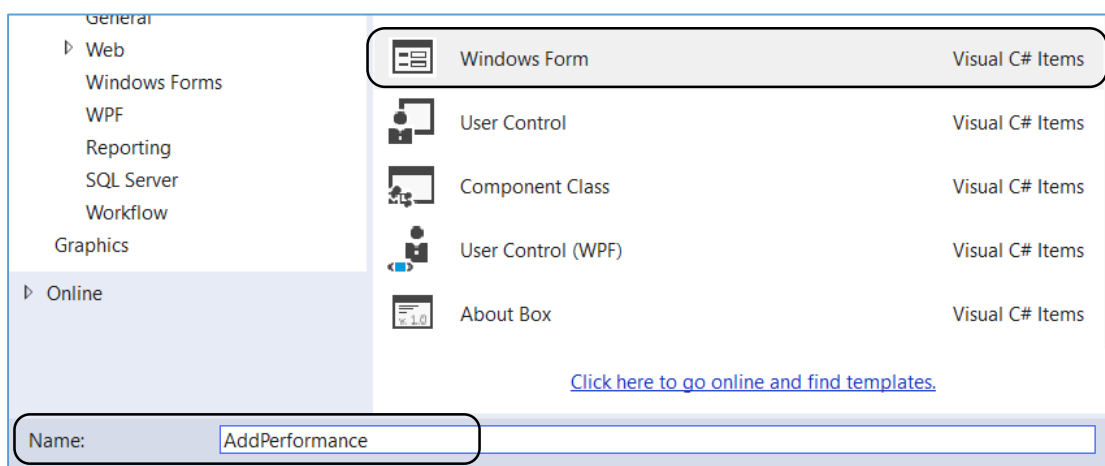
```

We can now run the program to check that the event text and '**Book seats**' buttons are displayed.



Before working on the theatre bookings, we must produce a form for entering the seat prices, dates and times of performances.

Add a **Windows Form** to the project. Give this the name '**AddPerformance**'.



Set up components on the form as shown:

The screenshot shows a Windows form titled "AddPerformance". The form contains the following components and labels:

- comboBox** *combEvents*: A dropdown menu labeled "Event" at the top.
- listBox** *listBox1*: A list box labeled "Performances" in the middle.
- dateTimePicker** *dateTimePicker1*: A date picker labeled "Date" showing "20 September 2015".
- textBox** *txtTime*: A text box labeled "Time" below the date picker.
- textBox** *txtPrice*: A text box labeled "Ticket price £" below the time text box.
- button** *btnAdd*: A button labeled "add performance" at the bottom left.
- button** *btnClose*: A button labeled "close" at the bottom right.

Dashed lines connect each label to its corresponding component on the form.

Link the **AddPerformance** form to the menu system by double clicking the 'Add performance' menu option on the **DisplayEvents** form, then add lines of code:

```
private void addPreformanceToolStripMenuItem_Click(object sender, EventArgs e)
{
    AddPerformance frmAddPerformance = new AddPerformance();
    frmAddPerformance.ShowDialog();
}
```

The first requirement for the **AddPerformance** form is to load a list of the theatre events into the drop down combo box.

Event objects have already been created when the program first runs, so there is no need to reload data from the database. We can simply use a loop to access the title from each eventObject and add this to the comboBox list.

Write a **loadEvents()** method for the **AddPerformance** form. Call this from the **AddPerformance()** method. Also add code to the 'close' button.

```
public AddPerformance()
{
    InitializeComponent();
    loadEvents();
}

private void loadEvents()
{
    string eventTitle;
    combEvents.Items.Clear();
    for (int i = 0; i < theatreEvent.eventCount; i++)
    {
        eventTitle = theatreEvent.eventObject[i].getTitle();
        combEvents.Items.Add(eventTitle);
    }
}

private void btnClose_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Run the program and check that the list of events is shown correctly in the comboBox list.

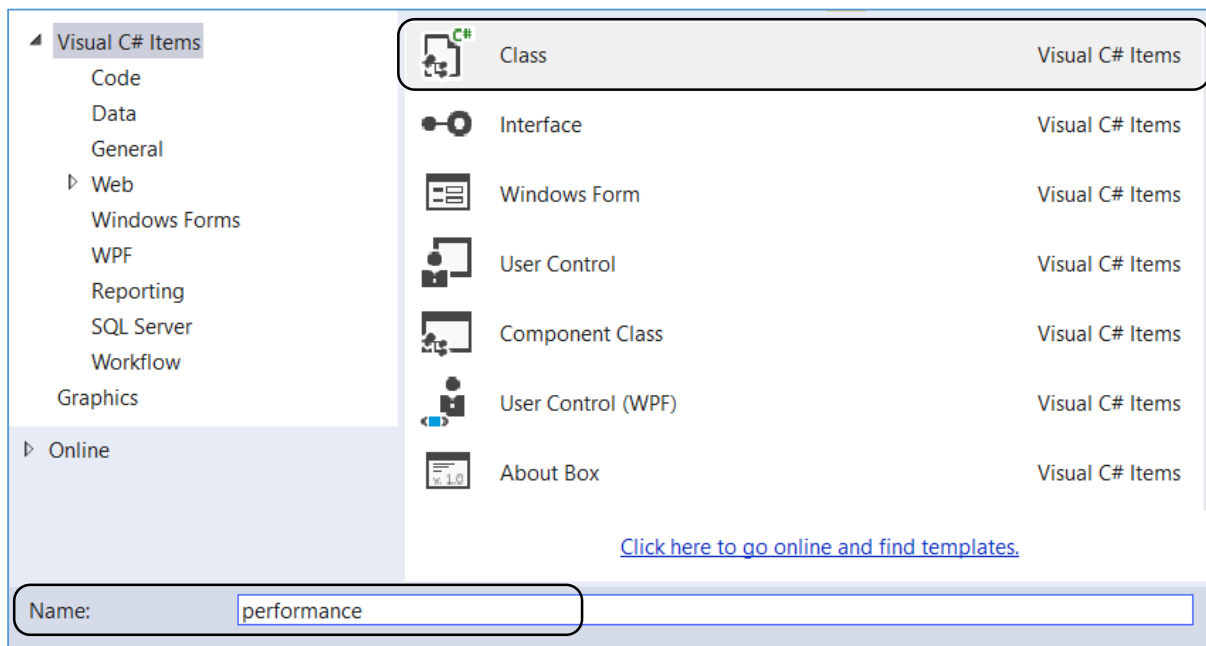
The screenshot shows a Windows Forms application window titled 'AddPerformance'. It contains the following controls:

- Event**: A label next to a dropdown menu. The dropdown is open, showing a list of events: 'Les Miserables', 'The Lion King', 'Jersey Boys', 'We Will Rock You' (highlighted in blue), 'Thriller', 'West Side Story', and 'Othello'.
- Performances**: A label next to an empty text box.
- Date**: A label next to a date picker showing '20 September 2015'.
- Time**: A label next to an empty text box.
- Ticket price £**: A label next to an empty text box.
- Buttons**: Two buttons at the bottom, 'add performance' and 'close'.

When the user clicks the '**add performance**' button, we want two things to happen:

- The **eventID**, **performance date**, **time** and **ticket price** should be saved into the **Performance** table of the database.
- A set of **seat** records will be created for the performance, all initially set as '**available for booking**'.

In preparation for these tasks, we will set up a class file called '**performance**'.



Add the properties for **performance** objects, and a set of methods for transferring data into and out of the property fields.

```
class performance
{
    private int performanceID;
    private int eventID;
    private DateTime performanceDate;
    private string time;
    private double seatPrice;

    public void setPerformanceID(int pID)
    {
        performanceID = pID;
    }

    public int getPerformanceID()
    {
        return performanceID;
    }

    public void setEventID(int e)
    {
        eventID = e;
    }
}
```

```

public int getEventID()
{
    return eventID;
}

public void setDate(DateTime d)
{
    performanceDate = d;
}

public DateTime getDate()
{
    return performanceDate;
}

public void setTime(string t)
{
    time = t;
}

public string getTime()
{
    return time;
}

public void setPrice(double p)
{
    seatPrice = p;
}

public double getPrice()
{
    return seatPrice;
}

```

We will now create a method to save a performance record into the database. It will be necessary to add '*using SqlClient*' and '*using Data*' directives, and to give the database location.

```

using System.Linq;
using System.Text;

using System.Data.SqlClient;
using System.Data;

namespace theatreBookings
{
    class performance
    {
        private static string databaseLocation = "C:\\C#\\theatreBookings.mdf;";
    }
}

```

Write the **AddPerformance()** method. Remember that **performanceID** is an auto-number generated by the computer. We will need to know this value, as it forms a property of the **seat** objects which will be created for the performance. We can obtain the **performanceID** for the record which has just been saved by using the '**SELECT SCOPE_IDENTITY()**' command.

```
private int performanceID;
private int eventID;
private DateTime performanceDate;
private string time;
private double seatPrice;

public static void AddPerformance(int ID, DateTime d, string t, double p)
{
    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;
        AttachDbFilename=" + databaseLocation + "Integrated Security=True;
        Connect Timeout=30; User Instance=True");
    con.Open();
    SqlCommand cmPerformance = new SqlCommand();
    cmPerformance.Connection = con;
    cmPerformance.CommandType = CommandType.Text;
    cmPerformance.CommandText = "INSERT INTO Performance(eventID, performanceDate,"
        + "time, seatPrice) VALUES ('" + ID + "', '" + d.ToString("MM/dd/yyyy") + "', '"
        + t + "', '" + p + "')";
    cmPerformance.ExecuteNonQuery();

    cmPerformance.CommandText = "SELECT SCOPE_IDENTITY()";
    int identity = Convert.ToInt32(cmPerformance.ExecuteScalar());
    con.Close();
    assignSeats(identity);
}
```

Add the **assignSeats()** method to create the set of seat objects for the performance.

```
public static void assignSeats(int performanceID)
{
    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;
        AttachDbFilename=" + databaseLocation + "Integrated Security=True;
        Connect Timeout=30; User Instance=True");
    con.Open();
    SqlCommand cmSeat = new SqlCommand();
    cmSeat.Connection = con;
    cmSeat.CommandType = CommandType.Text;
    for (int r = 1; r <= 11; r++)
    {
        char rowLetter = Convert.ToChar(64 + r);
        if (r > 8)
            rowLetter = Convert.ToChar(65 + r);
        for (int s = 1; s <= 20; s++)
        {
            cmSeat.CommandText = "INSERT INTO Seat(seatRow, seatNumber, "
                + "performanceID, available, bookingID) VALUES ('" + rowLetter + "', '"
                + s + "', '" + performanceID + "', '" + "0" + "', '" + "0" + "')";
            cmSeat.ExecuteNonQuery();
        }
    }
    con.Close();
}
```


This method creates a block of 11 rows of 20 seats. Some rows in the theatre have fewer than 20 seats, but the additional records can just be ignored and will not be accessed by the booking system.

Notice how the row number is converted to a letter using ASCII code: letter 'A' has ASCII value 65, 'B' is 66, etc. One slight complication is that the theatre does not use a row letter 'I', going instead from row 'H' to row 'J'. This is common practice, to avoid confusion between the letter 'I' the number 1. We compensate for the missing letter by altering the ASCII code calculation for row numbers above 8.

Return to the **AddPerformance** form and double click the '**add performance**' button. Add code to the button click method which will gather the necessary information for a new performance, then send this to the **AddPerformance()** method of the **performance** class.

```
private void btnAdd_Click(object sender, EventArgs e)
{
    DateTime performanceDate = Convert.ToDateTime(dateTimePicker1.Value);
    string performanceTime = txtTime.Text;
    double seatPrice = Convert.ToDouble(txtPrice.Text);
    int i = combEvents.SelectedIndex;
    int eventID = theatreEvent.eventObject[i].getEventID();
    performance.AddPerformance(eventID, performanceDate, performanceTime,
                              seatPrice);
    this.Close();
}
```

Run the program and enter test data for performances of different events. When each entry is complete, click the '**add performance**' button.

The screenshot shows a Windows application window titled "AddPerformance". Inside the window, there is a form with the following elements:

- Event:** A dropdown menu showing "Othello".
- Performances:** A label followed by a large, empty rectangular box for listing performances.
- Date:** A date picker showing "09 October 2015".
- Time:** A text box containing "19:00".
- Ticket price £:** A text box containing "18.50".
- Buttons:** Two buttons at the bottom, "add performance" and "close".

Exit from the program and go to the Server Explorer. Check that the performances you have entered are stored correctly in the **Performance** table. Check that the **eventID** corresponds with the correct theatre event.

performan...	eventID	performanceDate	time	seatPrice
1	7	09/10/2015 00:00:00	19:00	18.50
2	7	10/10/2015 00:00:00	19:00	21.50
3	5	25/09/2015 00:00:00	18:30	19.00
4	4	24/10/2015 00:00:00	18:30	17.50
5	2	17/10/2015 00:00:00	19:00	32.50
6	2	18/10/2015 00:00:00	19:00	32.50
7	3	27/10/2015 00:00:00	18:30	26.80
8	NULL	NULL	NULL	NULL

Go now to the **Seat** table. A set of 11 rows (A-L) of 20 seats should have been created for each performance. The 'available' field of each seat record will be set to 0, indicating that the seat has not yet been booked.

seatRow	seatNumber	performanceID	available	bookingID
L	13	1	0	0
L	14	1	0	0
L	15	1	0	0
L	16	1	0	0
L	17	1	0	0
L	18	1	0	0
L	19	1	0	0
L	20	1	0	0
A	1	2	0	0
A	2	2	0	0
A	3	2	0	0
A	4	2	0	0
A	5	2	0	0
A	6	2	0	0
A	7	2	0	0
A	8	2	0	0
A	9	2	0	0
A	10	2	0	0
A	11	2	0	0
A	12	2	0	0
A	13	2	0	0
A	14	2	0	0
A	15	2	0	0
A	16	2	0	0
A	17	2	0	0
A	18	2	0	0
A	19	2	0	0
A	20	2	0	0
B	1	2	0	0
B	2	2	0	0
B	3	2	0	0

One additional feature which we will include on the **AddPerformance** form is a list of the performances which have already been entered for each event. This information will be displayed in the **listBox**.

Begin by going to the **performance** class file and adding a **loadPerformances()** method. This takes as a parameter the **eventID** of the required theatre event, then searches the **performance** table in the database for any performances of this event.

When performance records have been found, these are used to create a set of **performance objects**.

We must also add variables to record the number of performance objects created, and an array to hold these objects.

```
private int performanceID;
private int eventID;
private DateTime performanceDate;
private string time;
private double seatPrice;

public static int performanceCount;
public static performance[] performanceObject = new performance[12];

public static void loadPerformances(int e)
{
    performanceCount = 0;
    DataSet dsPerformances = new DataSet();

    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;
        AttachDbFilename=" + databaseLocation + "Integrated Security=True;
        Connect Timeout=30; User Instance=True");
    con.Open();
    SqlCommand cmPerformances = new SqlCommand();
    cmPerformances.Connection = con;
    cmPerformances.CommandType = CommandType.Text;
    cmPerformances.CommandText = "SELECT * FROM Performance WHERE eventID='"
        + e + "'";
    SqlDataAdapter daPerformances = new SqlDataAdapter(cmPerformances);
    daPerformances.Fill(dsPerformances);
    con.Close();

    performanceCount = dsPerformances.Tables[0].Rows.Count;
    for (int i = 0; i < performanceCount; i++)
    {
        performanceObject[i] = new performance();
        DataRow dataRow = dsPerformances.Tables[0].Rows[i];
        performanceObject[i].setPerformanceID((int)dataRow[0]);
        performanceObject[i].setEventID((int)dataRow[1]);
        performanceObject[i].setDate(Convert.ToDateTime(dataRow[2]));
        performanceObject[i].setTime(Convert.ToString(dataRow[3]));
        performanceObject[i].setPrice(Convert.ToDouble(dataRow[4]));
    }
}
```

Return to the **AddPerformance** form. Double click the events comboBox to create an **indexchanged()** method. This will operate whenever the user selects a different option from the drop down list.

Add code to the method. This calls the **loadPerformances()** method in the **performance** class file, using **eventID** to specify which theatre event has been selected. Objects are created for all performances of the required event. We then use a loop to access each **performance object** and display the information in the listBox.

```
private void combEvents_SelectedIndexChanged(object sender, EventArgs e)
{
    listBox1.Items.Clear();
    int i=combEvents.SelectedIndex;
    int eventID = theatreEvent.eventObject[i].getEventID();
    try
    {
        performance.loadPerformances(eventID);
    }
    catch
    {
        MessageBox.Show("File error");
    }
    for (i = 0; i < performance.performanceCount; i++)
    {
        DateTime performanceDate = performance.performanceObject[i].getDate();
        string format = "ddd d MMM yyyy";
        string performanceDateString = performanceDate.ToString(format);
        listBox1.Items.Add("Date: " + performanceDateString);
        string performanceTime = performance.performanceObject[i].getTime();
        listBox1.Items.Add("Time: " + performanceTime);
        double seatPrice = performance.performanceObject[i].getPrice();
        string seatPriceString = seatPrice.ToString("f2");
        listBox1.Items.Add("Seat price: £" + seatPriceString);
        listBox1.Items.Add("");
    }
}
```

Run the program. Select an event, and details of all previously entered performances for that event should be displayed.

The screenshot shows a Windows application window titled 'AddPerformance'. At the top, there is a label 'Event' next to a dropdown menu currently showing 'Othello'. Below this is a label 'Performances' next to a list box. The list box contains two lines of text: 'Date: Fri 9 Oct 2015, Time: 19:00, Seat price: £18.50' and 'Date: Sat 10 Oct 2015, Time: 19:00, Seat price: £21.50'. Below the list box are three input fields: 'Date' with a calendar icon and the text '20 September 2015', 'Time' with an empty text box, and 'Ticket price £' with an empty text box.

We are now ready to start work on the theatre plan screen which will allow the user to select the seats for a booking.

Set up a new Windows Form with the name '*TheatrePlan*'. Return to the *DisplayEvents* form and add code to the empty *loadPlan()* method which you created earlier on page 177.

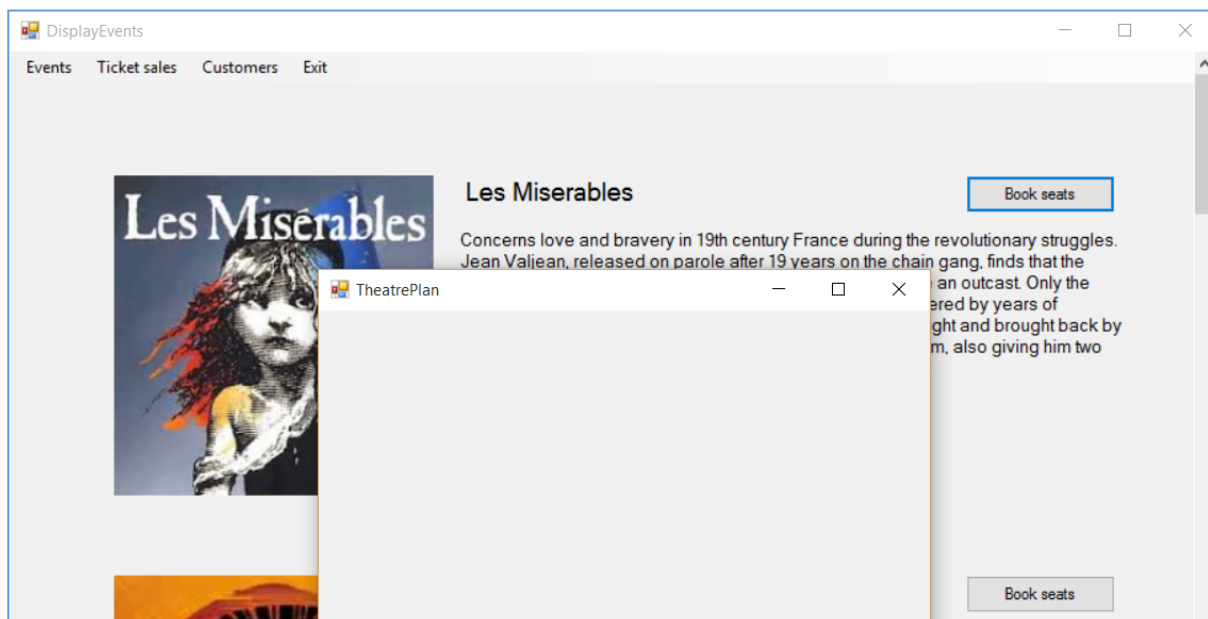
```
private void loadPlan(object sender, EventArgs e)
{
    Button clickedButton = (Button)sender;
    string s = clickedButton.Name;
    int i = Convert.ToInt16(s.Substring(3, 1));
    TheatrePlan frmTheatrePlan = new TheatrePlan();
    frmTheatrePlan.showTheatre(i);
    frmTheatrePlan.ShowDialog();
}
```

Go to the *TheatrePlan* form and add an empty *showTheatre()* method. This will bring in a parameter 'i' to indicate which theatre event had been selected from the scrolling list on the *DisplayEvents* page.

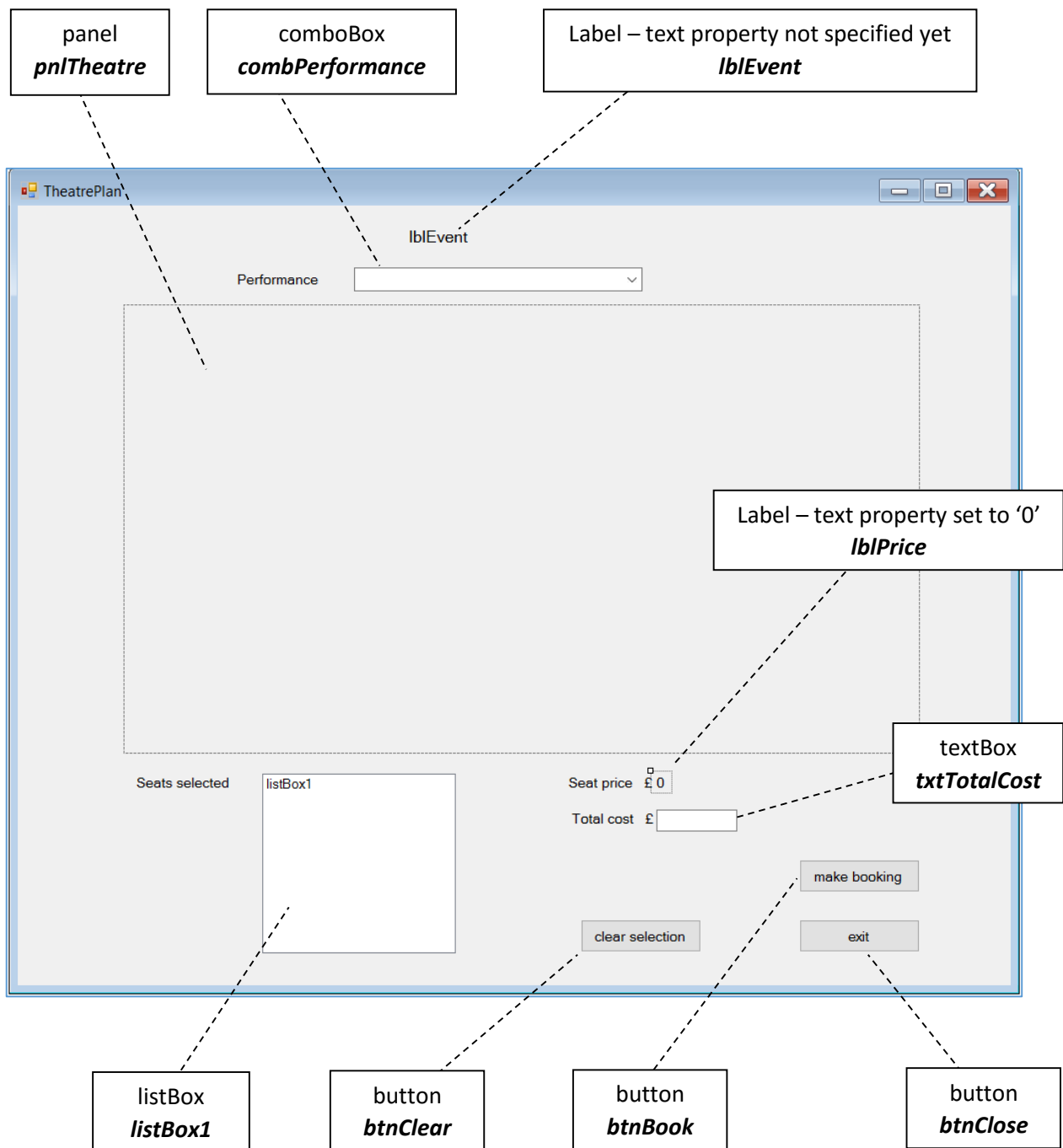
```
public TheatrePlan()
{
    InitializeComponent();
}

public void showTheatre(int i)
{
}
```

Run the program. Click one of the '*Book seats*' buttons and check that the *TheatrePlan* form opens correctly.



Set up components on the TheatrePlan form as show below:



Add code to the 'exit' button:

```
private void btnClose_Click(object sender, EventArgs e)
{
    this.Close();
}
```

The first requirement of the booking screen is to display the title of the selected theatre event, and provide a drop down list of the performance dates and times in the comboBox. Add code to the **showTheatre()** method to do this. Notice how this reuses the **loadPerformances()** method which we wrote earlier in the **performance** class. Add a definition for the **eventTitle** variable above the method heading.

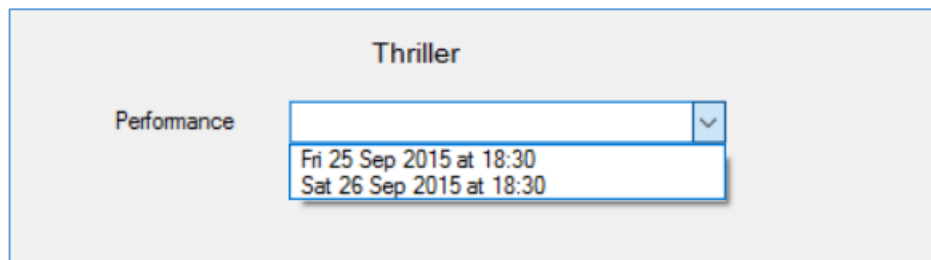
```
string eventTitle;

public void showTheatre(int i)
{
    eventTitle = theatreEvent.eventObject[i].getTitle();
    lblEvent.Text = eventTitle;

    int eventID = theatreEvent.eventObject[i].getEventID();
    performance.loadPerformances(eventID);

    combPerformance.Items.Clear();
    for (int p = 0; p < performance.performanceCount; p++)
    {
        DateTime performanceDate = performance.performanceObject[p].getDate();
        string format = "ddd d MMM yyyy";
        string performanceDateString = performanceDate.ToString(format);
        string performanceTime = performance.performanceObject[p].getTime();
        combPerformance.Items.Add(performanceDateString + " at "
                                   + performanceTime);
    }
}
```

Run the program. Select an event and check that the title and performance dates/times are displayed correctly.



We can now start to construct the seating plan diagram for the theatre. It is first necessary to make a small graphics image to represent a theatre seat on the plan. This should be about 20 pixels square, and can be saved in .PNG format with the name '**seat1.png**'



Go to the SolutionExplorer window and right click on the **theatreBookings** program icon. Load the seat image into the C# project by selecting '**Add / Existing item**', then locating the **seat1.png** graphics file in the file selection window.

We will build up the theatre plan from buttons displaying the seat image, using a similar technique to the *Solitaire* board display in chapter 4. Add an array at the start of the *TheatrePlan* form to hold these buttons.

```
public partial class TheatrePlan : Form
{
    Button[,] btnSeat = new Button[22, 12];

    public TheatrePlan()
    {
        InitializeComponent();
    }
}
```

You may recall from the program specification that the theatre has quite a complicated arrangement of seats:

				1	2	3	4	5	6	7	8	9	10	11	A	12	13	14		
			1	2	3	4	5	6	7	8	9	10	11	12	B	13	14	15	16	
		1	2	3	4	5	6	7	8	9	10	11	12	13	C	14	15	16	17	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	D	15	16	17	18	19
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	E	16	17	18	19	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	F	16	17	18	19	20
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	G	15	16	17	18	19
		1	2	3	4	5	6	7	8	9	10	11	12	13	H	15	16	17	18	19
			1	2	3	4	5	6	7	8	9	10	11	12	J	15	16	17	18	19
				1	2	3	4	5	6	7	8	9	10	11	K	16	17	18	19	20
					1	2	3	4	5	6	7	8	9	10	L	16	17	18	19	

We will construct this plan in several stages. The first will be to display the correct number of seats in each row. Create a ***drawPlan()*** method in the ***TheatrePlan*** form.

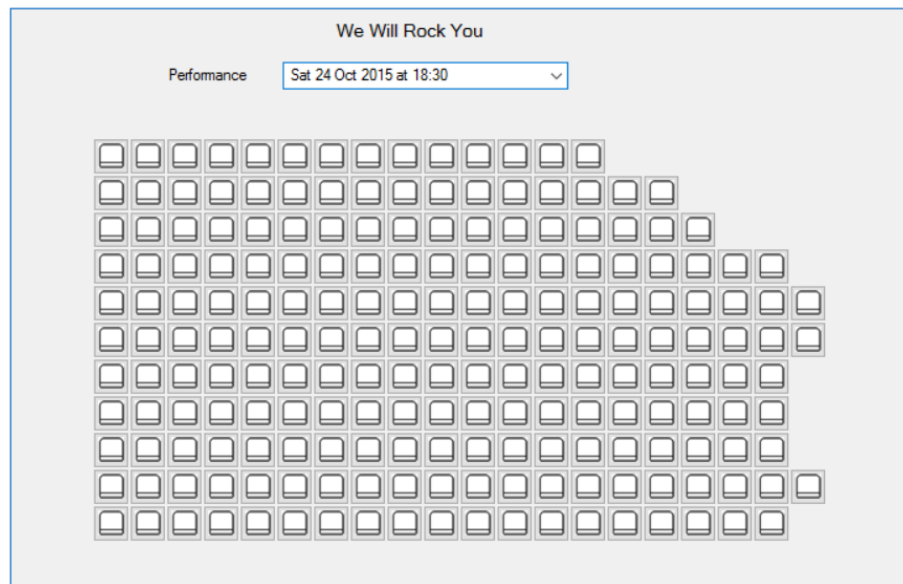
```
private void drawPlan()  
{  
    int seatMax;  
    for (int j = 1; j <= 11; j++)  
    {  
        seatMax = 20;  
        if (j == 1) seatMax = 14;  
        if (j == 2) seatMax = 16;  
        if (j == 3) seatMax = 17;  
        if (j == 4) seatMax = 19;  
        if (j >= 7 && j <= 9) seatMax = 19;  
        if (j == 11) seatMax = 19;  
        for (int i = 1; i <= seatMax; i++)  
        {  
            btnSeat[i, j] = new Button();  
            btnSeat[i, j].Width = 28;  
            btnSeat[i, j].Height = 28;  
            btnSeat[i, j].Left = (28 * i);  
            btnSeat[i, j].Top = (28 * j);  
            btnSeat[i, j].Image = Image.FromFile(".././seat1.png");  
            pnlTheatre.Controls.Add(btnSeat[i, j]);  
        }  
    }  
}
```


This method uses an outer loop to repeat for each of the 11 rows of seats, and an inner loop to repeat for the seats along each row. Before running the inner loop, the maximum number of seats is specified according to the theatre plan – this may vary between 14 and 20, depending on the row. The inner loop then creates button objects displaying the seat image.

Double click the comboBox component, and add a line of code to call the ***drawPlan()*** method when a performance date is selected.

```
private void combPerformance_SelectedIndexChanged(object sender, EventArgs e)
{
    drawPlan();
}
```

Run the program. Select an event and click the 'Book seats' button, then choose a performance date/time from the comboBox on the ***TheatrePlan*** form. The array of seats should be displayed. Check that the correct number of seats are present in each row.



We are making progress, but you will notice from the seating plan that rows start at different distances from the left wall of the theatre. We must allow for this if our seating plan is to be realistic.

We can allocate a variable called '**offset**' which records the number of seat positions by which each row is indented:

offset = 4	<div></div>	<div></div>	<div></div>	<div></div>	1	2	3	4	5	6	7	8	9	10	11	A	12				
offset = 3	<div></div>	<div></div>	<div></div>		1	2	3	4	5	6	7	8	9	10	11	12	B	13			
offset = 2	<div></div>	<div></div>			1	2	3	4	5	6	7	8	9	10	11	12	13	C	14		
offset = 1	<div></div>				1	2	3	4	5	6	7	8	9	10	11	12	13	14	D	15	
offset = 0					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	E	16
offset = 0					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	F	16

Return to the **drawPlan()** method. Add the **offset** variable and the code to set the number of offset positions for each row of seats. Change the left positions of the buttons to allow for the offsets.

```
private void drawPlan()
{
    int offset;

    int seatMax;

    for (int j = 1; j <= 11; j++)
    {
        seatMax = 20;
        if (j == 1) seatMax = 14;
        if (j == 2) seatMax = 16;
        if (j == 3) seatMax = 17;
        if (j == 4) seatMax = 19;
        if (j >= 7 && j <= 9) seatMax = 19;
        if (j == 11) seatMax = 19;

        for (int i = 1; i <= seatMax; i++)
        {
            offset = 0;
            if (j == 1) offset = 4;
            if (j == 2) offset = 3;
            if (j == 3) offset = 2;
            if (j == 4) offset = 1;
            if (j >= 7 && j <= 9) offset = 1;

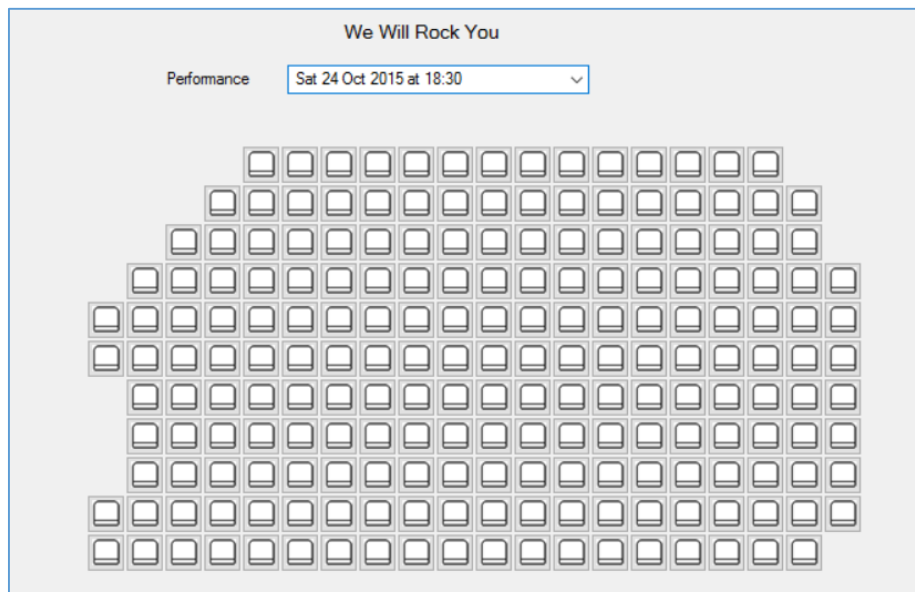
            btnSeat[i, j] = new Button();
            btnSeat[i, j].Width = 28;
            btnSeat[i, j].Height = 28;

            btnSeat[i, j].Left = ((28 * i) + (28 * offset));

            btnSeat[i, j].Top = (28 * j);
        }
    }
}
```

change this line

Run the program and check that the rows are now indented in the correct pattern.



The next slight complication is that the theatre has an aisle between the blocks of seats. On the plan, this can be used to display the row letters.

		1	2	3	4	5	6	7	8	9	10	11	A	12	13	14	
	1	2	3	4	5	6	7	8	9	10	11	12	B	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	C	14	15	16	17

We will need an array to hold the label which display the row letters. Add this at the start of *TheatrePlan*.

```
public partial class TheatrePlan : Form
{
    Button[,] btnSeat = new Button[22, 12];

    Label[] label = new Label[12];

    public TheatrePlan()
    {
        InitializeComponent();
    }
}
```

Add a line of code to the ***drawPlan()*** method which will move all seat positions to the right by 28 pixels if they are beyond the position of the aisle. This will create the gap between the seat blocks. We then add code to place labels in this gap to show the row letters.

Notice how the row letters are generated from the row numbers using **ASCII values**, and how the letter is incremented beyond row 8 to allow for the missing letter 'i'.

```

        btnSeat[i, j] = new Button();
        btnSeat[i, j].Width = 28;
        btnSeat[i, j].Height = 28;
        btnSeat[i, j].Left = ((28 * i) + (28 * offset));

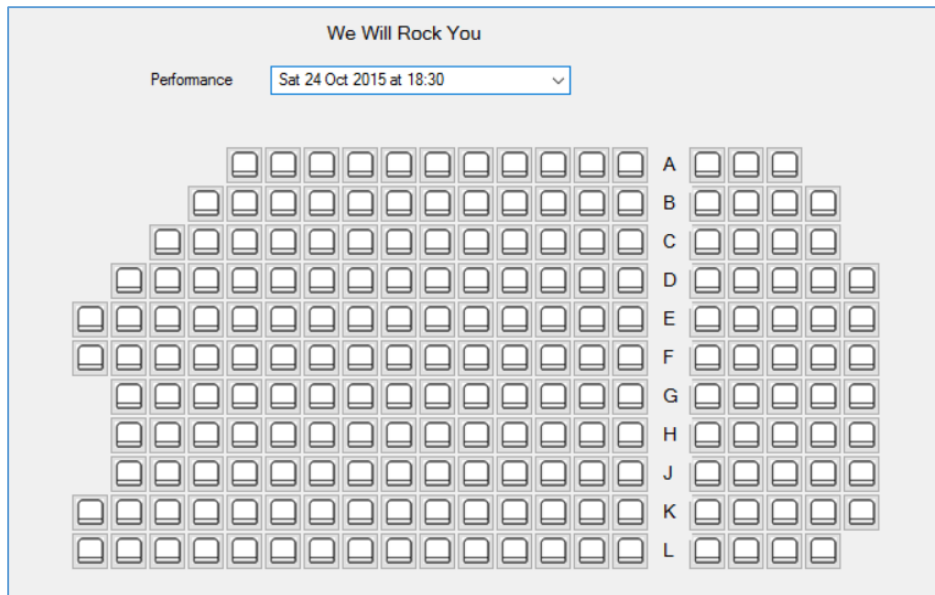
        if ((i + offset) > 15)
            btnSeat[i, j].Left += 28;

        btnSeat[i, j].Top = (28 * j);
        btnSeat[i, j].Image = Image.FromFile("../..\\seat1.png");
        pnlTheatre.Controls.Add(btnSeat[i, j]);
    }

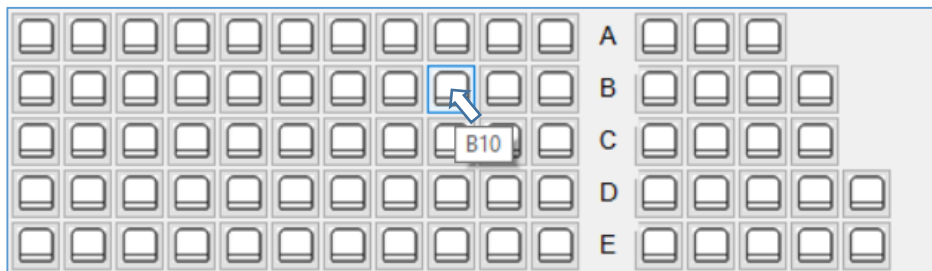
    label[j] = new Label();
    label[j].Size = new System.Drawing.Size(24, 30);
    char c = Convert.ToChar(64 + j);
    if (j > 8)
        c = Convert.ToChar(65 + j);
    label[j].Text = Convert.ToString(c);
    label[j].Font = new System.Drawing.Font("Microsoft Sans Serif",
        10F, System.Drawing.FontStyle.Regular,
        System.Drawing.GraphicsUnit.Point, ((byte)(0)));
    label[j].Location = new System.Drawing.Point(455, 4 + 28 * j);
    pnlTheatre.Controls.Add(label[j]);
}
}

```

Run the program and select an event and performance. The theatre plan should now be displayed with an aisle and row letters.



A further small feature that you might like to implement is a **toolTip** message box. This is a small box which appears when the user hovers the mouse over a component. We can set this up to display the seat row and number:



Return to the **drawPlan()** method and add code to link a tooltip message to each seat button. We again make use of ASCII codes in converting the row number to the corresponding row letter.

```

btnSeat[i, j].Image = Image.FromFile(".././seat1.png");
pnlTheatre.Controls.Add(btnSeat[i, j]);

int rowNumber = j;
if (j > 8)
    rowNumber++;
string tooltipText = Convert.ToChar(rowNumber + 64).ToString()
    + (i).ToString();
ToolTip buttonToolTip = new ToolTip();
buttonToolTip.SetToolTip(btnSeat[i, j], tooltipText);
}
label[j] = new Label();
label[j].Size = new System.Drawing.Size(24, 30);
char c = Convert.ToChar(64 + j);

```

Run the program to check that the **toolTip** messages are displayed correctly.

We can now turn our attention to displaying the seat bookings which have been made for performances. To provide test data, go to the **Server Explorer**, connect the database and open the **Seat** table. Locate the set of seats for a performance and set some of the seats in **row A** as **booked**. This is done by changing the value of the 'available' field from **0** to **1**. Make a note of the event and performance to which these seats are allocated. When you have done this, close the table window and **delete the connection** to the database.

seatRow	seatNumber	performanceID	available	bookingID
A	1	2	0	0
A	2	2	0	0
A	3	2	1	0
A	4	2	1	0
A	5	2	1	0
A	6	2	1	0
A	7	2	0	0
A	8	2	0	0

It will be necessary to create a second version of the seat image, this time in a colour such as red, to represent a booked seat. Call this '**seat2.png**' and load it into the project using the '**Add / Existing item**' option:

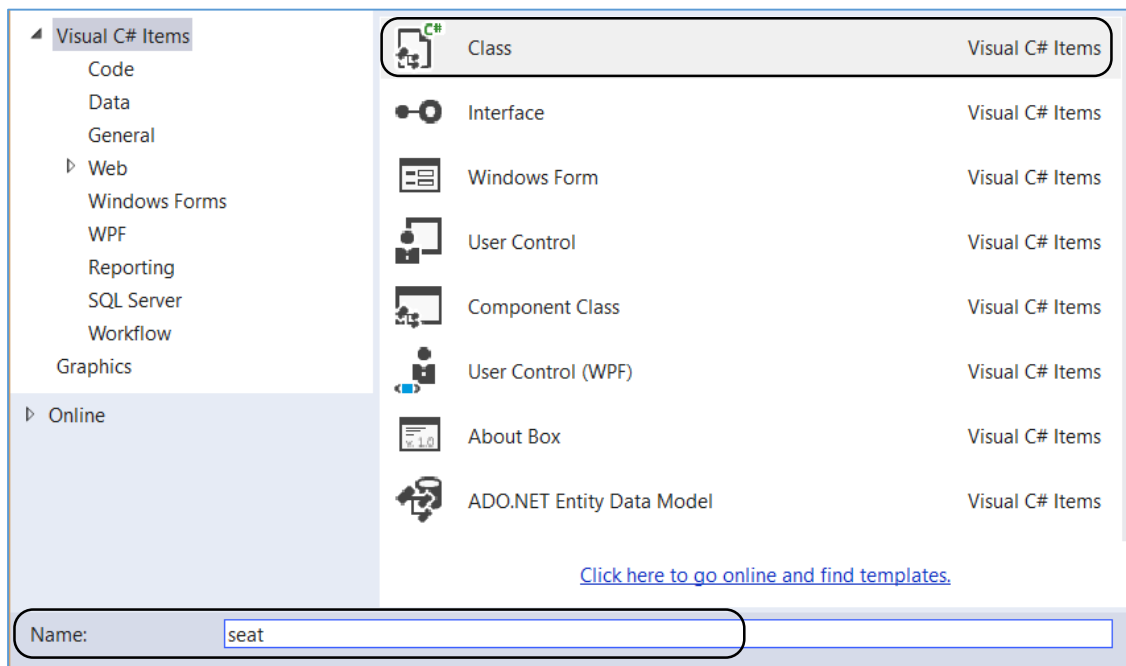


seat1.png



seat2.png

Before displaying seat bookings, we must create a **seat** object class to handle the data. Select '**Add / New item**' and create a class file called '**seat**'.



As with previous classes, insert the property fields for seat objects and the methods for transferring data into and out from these properties.

```
class seat
{
    private char seatRow;
    private int seatNumber;
    private int performanceID;
    private int available;
    private int bookingID;

    public void setSeatRow(char r)
    {
        seatRow = r;
    }

    public char getSeatRow()
    {
        return seatRow;
    }

    public void setSeatNumber(int s)
    {
        seatNumber = s;
    }

    public int getSeatNumber()
    {
        return seatNumber;
    }

    public void setPerformanceID(int p)
    {
        performanceID = p;
    }

    public int getPerformanceID()
    {
        return performanceID;
    }

    public void setAvailable(int a)
    {
        available = a;
    }

    public int getAvailable()
    {
        return available;
    }

    public void setBookingID(int b)
    {
        bookingID = b;
    }

    public int getBookingID()
    {
        return bookingID;
    }
}
```

We will now produce a ***loadSeats()*** method which uses a parameter '***p***' to specify the ***performanceID*** of the performance for which the seat data is required. The set of seat records is loaded from the database table, then used to create a set of seat objects.

It will be necessary to add '***using SqlClient***' and '***using Data***' directives to the start of the seat class file, set up a variable ***seatCount*** to record the number of seat objects created, and to declare an array to link to the seat objects.

```
using System.Linq;
using System.Text;

using System.Data.SqlClient;
using System.Data;

namespace theatreBookings
{
    class seat
    {
        private char seatRow;
        private int seatNumber;
        private int performanceID;
        private int available;
        private int bookingID;

        private static string databaseLocation="C:\\C#\\theatreBookings.mdf;";

        public static int seatCount;
        public static seat[] seatObject = new seat[240];

        public static void loadSeats(int p)
        {
            seatCount = 0;
            DataSet dsSeats = new DataSet();
            SqlConnection con = new SqlConnection(@"Data Source=.\\SQLEXPRESS;
            AttachDbFilename=" + databaseLocation + "Integrated Security=True;
            Connect Timeout=30; User Instance=True");
            con.Open();
            SqlCommand cmSeats = new SqlCommand();
            cmSeats.Connection = con;
            cmSeats.CommandType = CommandType.Text;
            cmSeats.CommandText = "SELECT * FROM Seat WHERE performanceID='"
                + p + "'";
            SqlDataAdapter daSeats = new SqlDataAdapter(cmSeats);
            daSeats.Fill(dsSeats);
            con.Close();

            seatCount = dsSeats.Tables[0].Rows.Count;
            for (int i = 0; i < seatCount; i++)
            {
                seatObject[i] = new seat();
                DataRow dataRow = dsSeats.Tables[0].Rows[i];
                seatObject[i].setSeatRow(Convert.ToChar(dataRow[0]));
                seatObject[i].setSeatNumber((int)dataRow[1]);
                seatObject[i].setPerformanceID((int)dataRow[2]);
                seatObject[i].setAvailable((int)dataRow[3]);
                seatObject[i].setBookingID((int)dataRow[4]);
            }
        }
    }
}
```

Return to the **TheatrePlan** form.

It will be convenient to use a two dimensional integer array, similar to the array in the Solitaire game, to keep track of the status of each seat as available or booked. Set up the **seatStatus** array at the start of **TheatrePlan**.

```
public partial class TheatrePlan : Form
{
    Button[,] btnSeat = new Button[22, 12];
    Label[] label = new Label[12];

    int[,] seatStatus = new int[21, 12];
}
```

Double click the comboBox to go to the **IndexChanged()** method. Add code which carries out several activities:

- The **performanceID** is first obtained from the selected performance date/time in the comboBox.
- The **loadSeats()** method in the **seat** class is called, which retrieves data from the database table and creates a set of **seat objects** for the required performance.
- A loop accesses each of the **seat** objects. The row letter is converted to a row number by means of its ASCII code.
- The value of the '**available**' property is stored in the **seatStatus** array using the seat and row numbers as the array indices '**s**' and '**r**'. The value of '**available**' will be 0 for an empty seat, and 1 for a booked seat.
- Existing seat buttons are removed from the panel, ready to redisplay the plan.
- Finally, the ticket price for the performance is displayed in the **lblPrice** label.

```
private void combPerformance_SelectedIndexChanged(object sender, EventArgs e)
{
    int p = combPerformance.SelectedIndex;
    int performanceID = performance.performanceObject[p].getPerformanceID();
    seat.loadSeats(performanceID);

    for (int i = 0; i < seat.seatCount; i++)
    {
        char c = seat.seatObject[i].getSeatRow();
        int r = ((int)c) - 64;
        if (r > 8)
            r--;
        int s = seat.seatObject[i].getSeatNumber();
        int available = seat.seatObject[i].getAvailable();
        seatStatus[s, r] = available;
    }

    int totalButtons = pnlTheatre.Controls.Count;
    for (int i = 0; i < totalButtons; i++)
    {
        pnlTheatre.Controls.RemoveAt(i);
    }
    lblPrice.Text = performance.performanceObject[p].getPrice().ToString("f2");

    drawPlan();
}
```

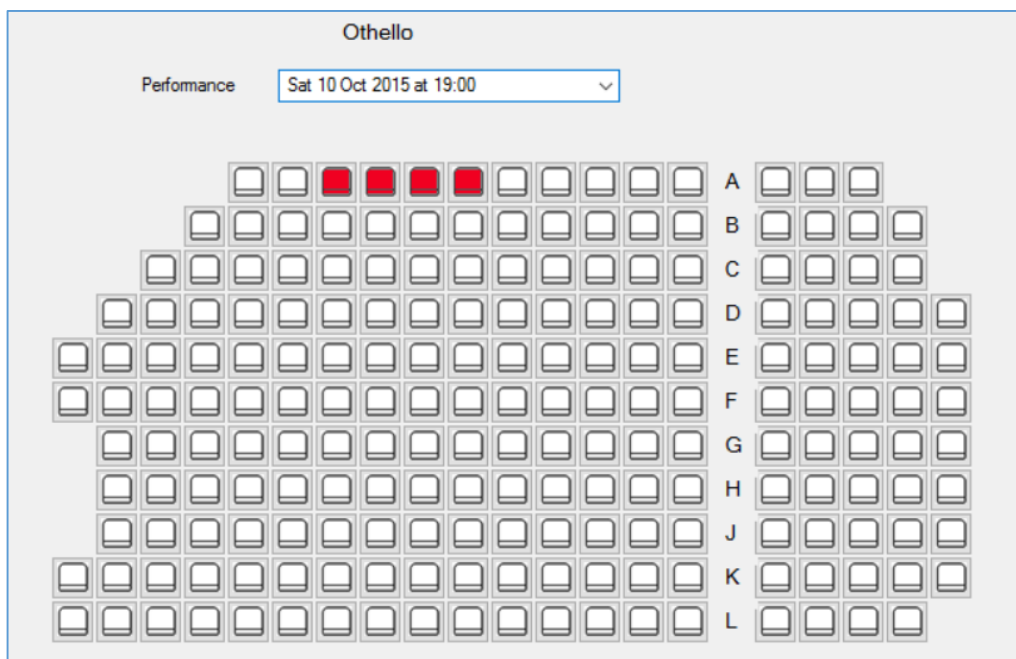

One change is needed to the **drawPlan()** method to display the coloured **seat2.png** image for booked seats, where **seatStatus** has a value of 1. Add the extra lines of code:

```
btnSeat[i, j] = new Button();
btnSeat[i, j].Width = 28;
btnSeat[i, j].Height = 28;
btnSeat[i, j].Left = ((28 * i) + (28 * offset));
if ((i + offset) > 15)
    btnSeat[i, j].Left += 28;
btnSeat[i, j].Top = (28 * j);
btnSeat[i, j].Image = Image.FromFile(".././seat1.png");

if (seatStatus[i, j] == 1)
    btnSeat[i, j].Image = Image.FromFile(".././seat2.png");

pnlTheatre.Controls.Add(btnSeat[i, j]);
```

Run the program. Select the event and performance for which you altered the '**available**' properties of some seats (page 195). The seats which you set as '**booked**' should appear with colour coding on the seat plan.



We can now move on to the task of selecting and booking seats. As the user selects seats by clicking on the theatre plan, it will be convenient for these seats to appear in another colour, for example: green. Make a third seat image, called '**seat3.png**' in this additional colour and load it into the project.



seat1.png



seat2.png



seat3.png

Return to the **drawPlan()** method and add code which will do two things:

- Each button is allocated a name, made up from the letters '**btn**', followed by the row number and the seat number. To avoid ambiguity, a space is added before a single digit row or seat number.
- A button click procedure called **seat_Click()** is linked to each button.

```

btnSeat[i, j].Image = Image.FromFile(".././seat1.png");
if (seatStatus[i, j] == 1)
    btnSeat[i, j].Image = Image.FromFile(".././seat2.png");

String buttonName = "btn";
if (j <= 9)
    buttonName += " ";
buttonName += j;
if (i <= 9)
    buttonName += " ";
buttonName += i;

btnSeat[i, j].Name = buttonName;
btnSeat[i, j].Click += new EventHandler(seat_Click);

pnlTheatre.Controls.Add(btnSeat[i, j]);

```

Add the **seatClick()** method immediately after **drawPlan()**. This will process the button click by carrying out a series of actions:

- The button name is broken down to obtain the row and seat number.
- The program checks that this seat does not have a **seatStatus** value of 1, which would indicate that it was already booked.
- If the seat is currently available, a **seatStatus** value of 2 is allocated, and the image is changed to **seat3.png**.
- If the seat is already selected, then the selection is cancelled by re-setting **seatStatus** to 0 and changing the image back to **seat1.png**.

```

private void seat_Click(object sender, EventArgs e)
{
    Button clickedButton = (Button)sender;

    string s = clickedButton.Name;
    int j = Convert.ToInt16(s.Substring(3, 2));
    int i = Convert.ToInt16(s.Substring(5, 2));

    if (seatStatus[i, j] != 1)
    {
        if (seatStatus[i, j] == 0)
        {
            seatStatus[i, j] = 2;
            btnSeat[i, j].Image = Image.FromFile(".././seat3.png");
        }
        else
        {
            seatStatus[i, j] = 0;
            btnSeat[i, j].Image = Image.FromFile(".././seat1.png");
        }
    }
}

```

Run the program and select your test event/performance. Check that seats can be selected by clicking on the seat plan, and that seats can be deselected by clicking a second time.



We can display details of the seats selected in the **listBox** at the bottom of the form. Add a **listSelectedSeats()** method to the **TheatrePlan** page below the **seatClick()** method.

```
private void listSelectedSeats()
{
    try
    {
        int p = combPerformance.SelectedIndex;
        double seatPrice = performance.performanceObject[p].getPrice();
        totalCost = 0;
        listBox1.Items.Clear();
        for (int j = 1; j <= 11; j++)
        {
            for (int i = 1; i <= 20; i++)
            {
                if (seatStatus[i, j] == 2)
                {
                    char c = Convert.ToChar(64 + j);
                    if (j > 8)
                        c = Convert.ToChar(65 + j);
                    listBox1.Items.Add("Row " + c + " Seat " + i);
                    totalCost += seatPrice;
                }
            }
        }
        txtTotalCost.Text = totalCost.ToString("f2");
    }
    catch
    {
        MessageBox.Show("A performance must be selected");
    }
}
```

This method uses two loops to check the **seatStatus** array value for each seat. If a value of 2 is found, indicating that the user has selected that seat, then the seat row letter and seat number are added to the list box. We also take the opportunity to calculate the total cost of the seats booked, using the seat price loaded earlier from the **performance** object.

Add a line of code at the end of the **seatClick()** method to call **listSelectedSeats()**. We also need to add the **totalCost** variable.

```

        if (seatStatus[i, j] == 0)
        {
            seatStatus[i, j] = 2;
            btnSeat[i, j].Image = Image.FromFile("../..//seat3.png");
        }
        else
        {
            seatStatus[i, j] = 0;
            btnSeat[i, j].Image = Image.FromFile("../..//seat1.png");
        }

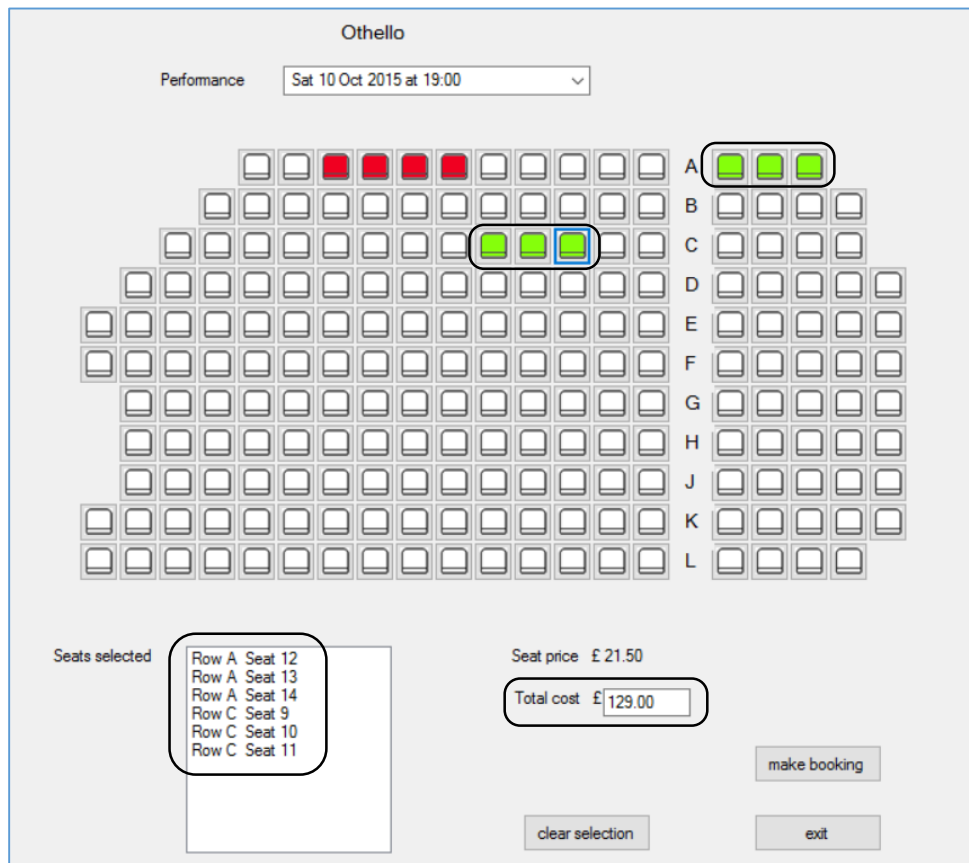
        listSelectedSeats();
    }
}

double totalCost;

private void listSelectedSeats()
{
    try
    {
        int p = combPerformance.SelectedIndex;
        double seatPrice = performance.performanceObject[p].getPrice();
        totalCost = 0;
        listBox1.Items.Clear();
    }
}

```

Run the program. Check that seats selected are added to the listBox, and that the total cost of the selected seats is calculated correctly.

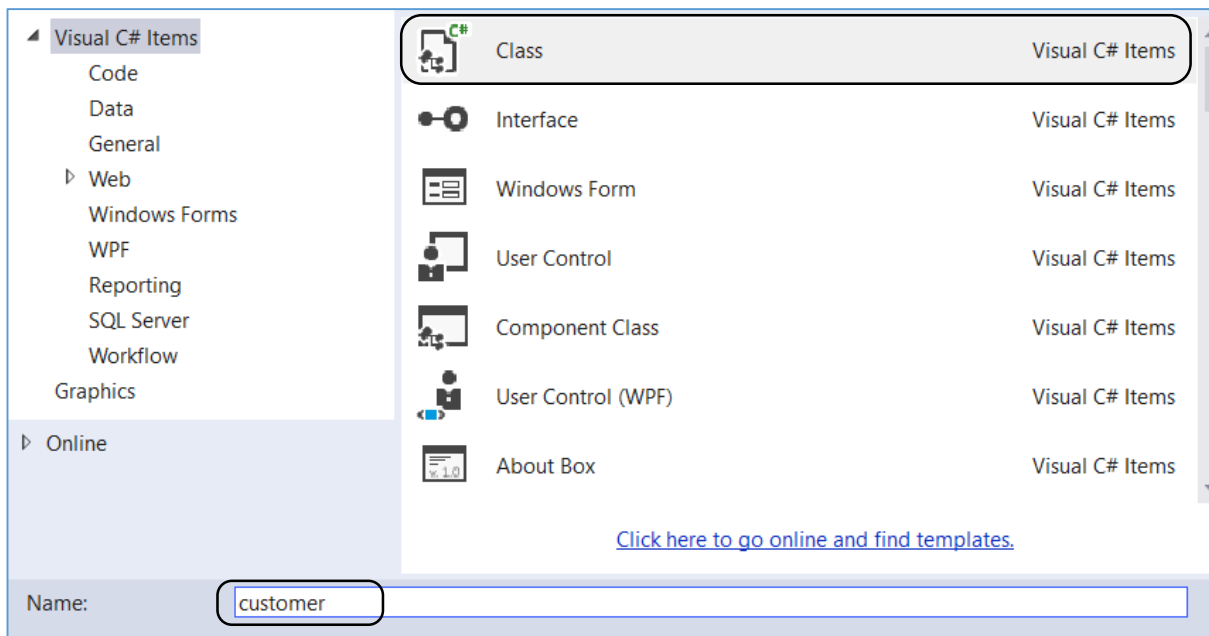


A final feature to complete is the '**clear selection**' button, which will reset all selected seats to the '**available**' state. Double click the button and add code to the button_Click method. This uses two loops to check the **seatStatus** array. If a value of 2 is found, indicating that the seat is currently selected, then the array value is reset to 0 and the image on the corresponding seat button is changed back to '**seat1.png**'.

```
private void btnClear_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();
    for (int j = 1; j <= 11; j++)
    {
        for (int i = 1; i <= 20; i++)
        {
            if (seatStatus[i, j] == 2)
            {
                seatStatus[i, j] = 0;
                btnSeat[i, j].Image = Image.FromFile("../..seat1.png");
            }
        }
    }
    txtTotalCost.Clear();
}
```

Run the program and check that seat selections can be cancelled.

Once the required seats have been selected, the user will move to the next form to input contact details for the customer. We will develop this section next, but first we must create a **customer** object class. Go to the **Solution Explorer** window and add a new class file with the name '**customer**'.



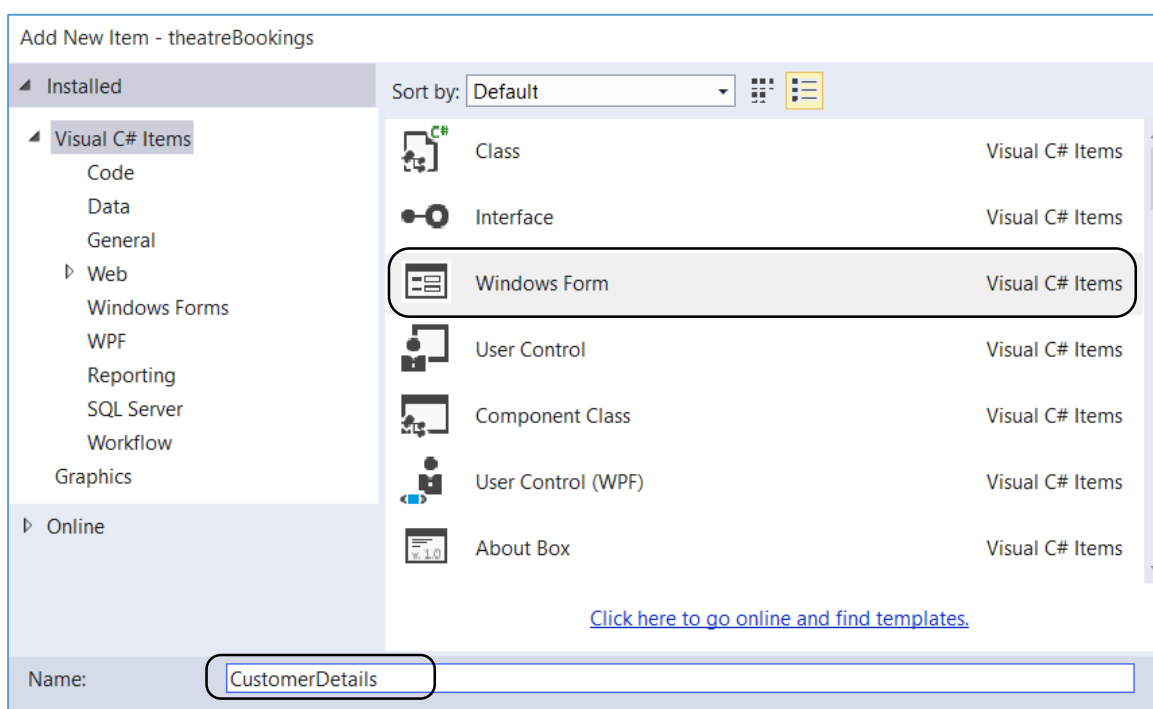
Add the properties for the **customer** class and the methods for transferring data into and out of **customer** objects, as shown below:

```
class customer
{
    private int customerID;
    private string surname;
    private string forename;
    private string title;
    private string address1;
    private string address2;
    private string town;
    private string postcode;
    private string email;
    private string phone;

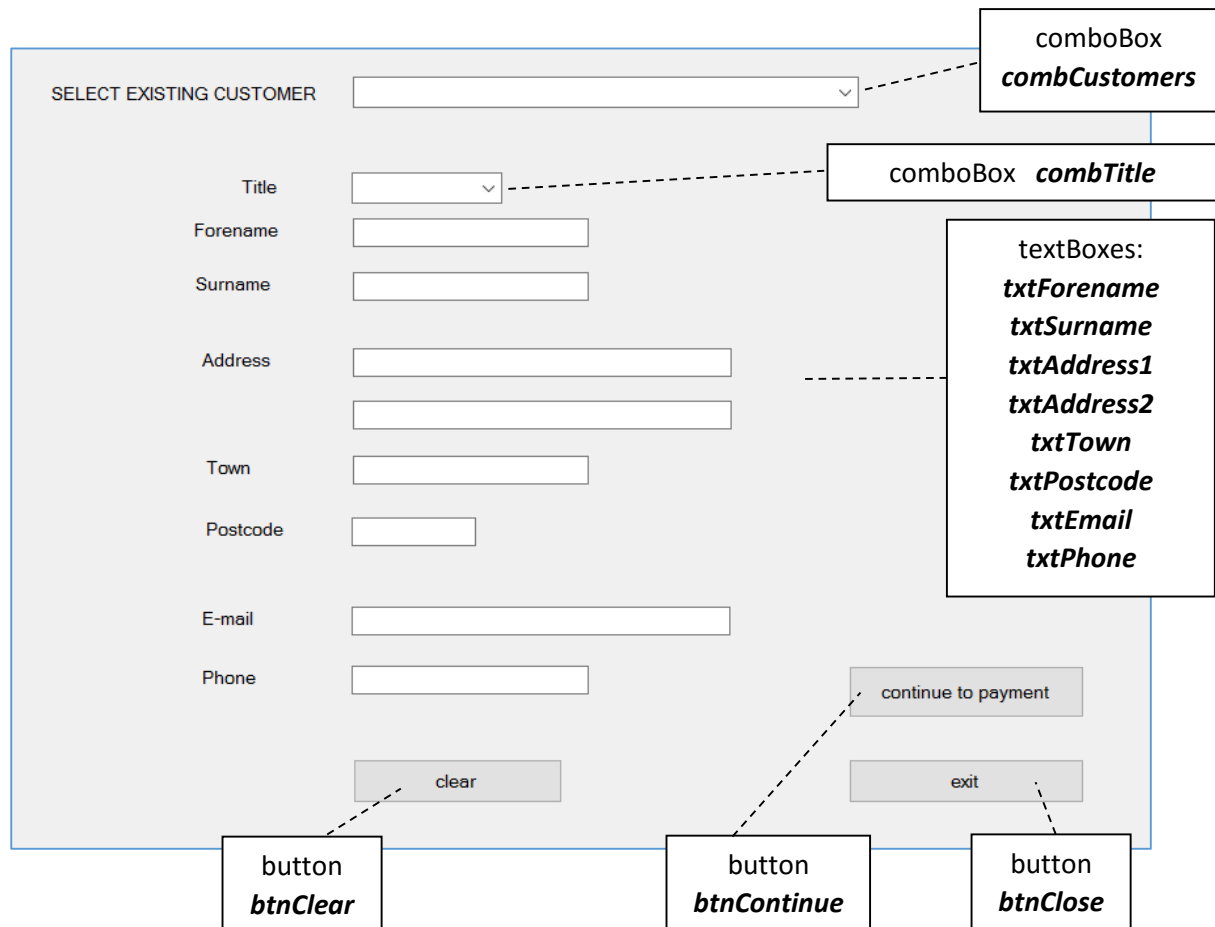
    public void setCustomerID(int c)
    {
        customerID = c;
    }
    public int getCustomerID()
    {
        return customerID;
    }
    public void setSurname(string s)
    {
        surname = s;
    }
    public string getSurname()
    {
        return surname;
    }
    public void setForename(string f)
    {
        forename = f;
    }
    public string getForename()
    {
        return forename;
    }
    public void setTitle(string t)
    {
        title = t;
    }
    public string getTitle()
    {
        return title;
    }
    public void setAddress1(string a1)
    {
        address1 = a1;
    }
    public string getAddress1()
    {
        return address1;
    }
    public void setAddress2(string a2)
    {
        address2 = a2;
    }
    public string getAddress2()
    {
        return address2;
    }
}
```

```
public void setTown(string to)
{
    town = to;
}
public string getTown()
{
    return town;
}
public void setPostcode(string pc)
{
    postcode = pc;
}
public string getPostcode()
{
    return postcode;
}
public void setEmail(string em)
{
    email = em;
}
public string getEmail()
{
    return email;
}
public void setPhone(string ph)
{
    phone = ph;
}
public string getPhone()
{
    return phone;
}
```

Return to the Solution Explorer window and add a Windows Form with the name **'CustomerDetails'**.



Add components to the **Customer Details** form:



We will need to carry forward various data from the **TheatrePlan** form to the **CustomerDetails** form, ready to record the booking in the database: the title of the theatre event, the performanceID, the seats selected and the total cost of the tickets. Go to the **CustomerDetails** form and add a **getBooking()** method and variables.

```
public partial class CustomerDetails : Form
{
    int f_performanceID;
    double f_totalCost;
    int[,] f_seatStatus;
    string f_eventTitle;

    public CustomerDetails()
    {
        InitializeComponent();
    }

    public void getBooking(int performanceID, double totalCost,
                          int[,] seatStatus, string eventTitle)
    {
        f_performanceID = performanceID;
        f_totalCost = totalCost;
        f_seatStatus = seatStatus;
        f_eventTitle = eventTitle;
    }
}
```


Add code to the **'exit'** button.

```
private void btnClose_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Return to the **TheatrePlan** form and double click the **'make booking'** button. Add code to the `btnBook_Click` method which will open the **CustomerDetails** form and call the **getBooking()** method, ready to transfer the required data.

```
private void btnBook_Click(object sender, EventArgs e)
{
    CustomerDetails frmCustomerDetails = new CustomerDetails();
    int p = combPerformance.SelectedIndex;
    int performanceID = performance.performanceObject[p].getPerformanceID();
    frmCustomerDetails.getBooking(performanceID, totalCost,
                                   seatStatus, eventTitle);
    frmCustomerDetails.ShowDialog();
    this.Close();
}
```

In the case of a **new customer**, the user will enter their name, address, e-mail and phone number. This information will then be saved into the **customer** table of the database. We will now write the method to save this record.

Go to the **customer class** file. Add **'using SqlClient'** and **'using Data'** directives, and insert the database location.

```
using System.Linq;
using System.Text;

using System.Data.SqlClient;
using System.Data;

namespace theatreBookings
{
    class customer
    {
        private static string databaseLocation = "C:\\C#\\theatreBookings.mdf;";

        private int customerID;
        private string surname;
        private string forename;
        private string title;
```

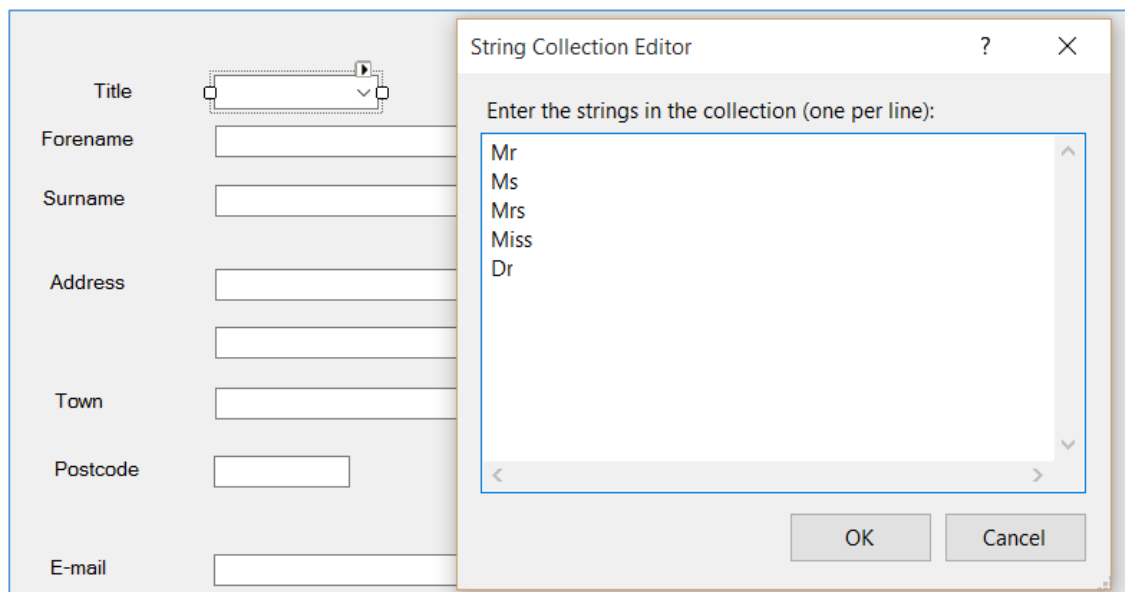
Add the **saveCustomer()** method to the **customer** class file. This takes as parameters the fields of the customer record, then uses the SQL **INSERT** command to save the record into the **customer** table of the database.

We will need to know the **customerID** value for the new record, so that this can be linked to the particular seat booking which is made. The computer will allocate a **customerID** automatically as an auto-number field. We can find the allocated value using the **SELECT SCOPE_IDENTITY()** command and return it from the **saveCustomer()** method.

```
public static int saveCustomer(string forename, string surname, string title,
    string address1, string address2, string town, string postcode, string email,
    string phone)
{
    SqlConnection con = new SqlConnection(@"Data Source=.\\SQLEXPRESS;
        AttachDbFilename=" + databaseLocation + "Integrated Security=True;
        Connect Timeout=30; User Instance=True");
    con.Open();
    SqlCommand cmCustomer = new SqlCommand();
    cmCustomer.Connection = con;
    cmCustomer.CommandType = CommandType.Text;
    cmCustomer.CommandText = "INSERT INTO Customer(forename,surname, title,"
        + "address1, address2, town, postcode, email, phone)"
        + "VALUES ('" + forename + "','" + surname + "','" + title + "','"
        + address1 + "','" + address2 + "','" + town + "','" + postcode + "','"
        + email + "','" + phone + "')";
    cmCustomer.ExecuteNonQuery();

    cmCustomer.CommandText = "SELECT SCOPE_IDENTITY()";
    int identity = Convert.ToInt32(cmCustomer.ExecuteScalar());
    con.Close();
    return identity;
}
```

Return to the **CustomerDetails** form. Select the comboBox for customer title. Go to the Properties window and select 'Items' to open the String Collection Editor window. Enter a choice of titles: Mr, Ms, Miss, Mrs, Dr...



Double click the '**continue to payment**' button and add code to the button_Click method. Also add a **customerID** variable immediately before the method.

The button_Click method first checks that data has been entered in the required fields: *surname*, *forename*, *address1* and *postcode*. If data is present, then all data fields are passed as parameters to the **saveCustomer()** method in the **customer** class, which saves the record to the database table.

```
int customerID;

private void btnContinue_Click(object sender, EventArgs e)
{
    customerID = 0;
    if (txtSurname.Text.Length > 0 && txtForename.Text.Length > 0 &&
        txtAddress1.Text.Length > 0 && txtPostcode.Text.Length > 0)
    {
        customerID = customer.saveCustomer(txtForename.Text, txtSurname.Text,
            combTitle.Text, txtAddress1.Text, txtAddress2.Text, txtTown.Text,
            txtPostcode.Text, txtEmail.Text, txtPhone.Text);
    }
    else
    {
        MessageBox.Show(
            "Missing customer information - name, address, postcode are required");
    }
}
```

Run the program. Select an event and performance, then click the '**make booking**' button to open the **CustomerDetails** form.

Enter test data for a customer. Click the '**continue to payment**' button, then '**exit**'.

The screenshot shows a Windows form titled 'CustomerDetails'. It contains several text boxes and a dropdown menu for entering customer information. The data entered is as follows:

Field	Value
Title	Ms
Forename	Denise
Surname	Adams
Address	Bay View
	27 Beach Road
Town	Tywyn
Postcode	LL29 8TN
E-mail	adams94@telecom.com
Phone	01672 345622

At the bottom of the form, there are three buttons: 'clear', 'continue to payment', and 'exit'. The 'continue to payment' button is highlighted with a blue border.

Enter several more customers in a similar way, then close the program windows. Go to the **Server Explorer** window and link to the **theatreBookings.mdf** database. Open the **customer** table and check that your test records have been saved, then delete the data connection.

customerID	forename	surname	title	address1	address2	town	postcode	email	phone
1	Denise	Adams	Ms	Bay View	27 Beach Road	Tywyn	LL29 8TN	adams94@telecom.com	01672 345622
2	Stuart	Andrews	Mr	Ty Gwyn	Harlech Road	Barmouth	LL39 7TR	andrews6@telecom.com	01762 345671
3	Ruth	Southhill	Ms	17 West Beach		Fairbourne	LL45 6TW	ruthsouth@btinternet.com	01567 341344
4	Judith	Williams	Mrs	36 Heol Glyndwr		Machynlleth	LL31 5UP	williams56@gmail.com	01678 965477
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Apart from entering new customers, we also want to be able to select details of existing customers from the database. Go to the customer class and add a **loadCustomers()** method, a **customerCount** variable and an **array** to link to customer objects.

The **loadCustomers()** method will load all customer records from the database. We ask for these to be sorted alphabetically by surname with the '**ORDER BY...**' command in SQL. A **customerObject** is then created from each of the records.

```
private string email;
private string phone;
private static string databaseLocation = "C:\\C#\\theatreBookings.mdf";

public static int customerCount;
public static customer[] customerObject = new customer[12];

public static void loadCustomers()
{
    customerCount = 0;
    DataSet dsCustomers = new DataSet();
    SqlConnection con = new SqlConnection(@"Data Source=.\\SQLEXPRESS;
        AttachDbFilename=" + databaseLocation + "Integrated Security=True;
        Connect Timeout=30; User Instance=True");
    con.Open();
    SqlCommand cmCustomers = new SqlCommand();
    cmCustomers.Connection = con;
    cmCustomers.CommandType = CommandType.Text;
    cmCustomers.CommandText = "SELECT * FROM Customer ORDER BY surname ASC";
    SqlDataAdapter daCustomers = new SqlDataAdapter(cmCustomers);
    daCustomers.Fill(dsCustomers);
    con.Close();

    customerCount = dsCustomers.Tables[0].Rows.Count;
    for (int i = 0; i < customerCount; i++)
    {
        customerObject[i] = new customer();
        DataRow dataRow = dsCustomers.Tables[0].Rows[i];
        customerObject[i].setCustomerID((int)dataRow[0]);
        customerObject[i].setForename(Convert.ToString(dataRow[1]));
        customerObject[i].setSurname(Convert.ToString(dataRow[2]));
        customerObject[i].setTitle(Convert.ToString(dataRow[3]));
        customerObject[i].setAddress1(Convert.ToString(dataRow[4]));
        customerObject[i].setAddress2(Convert.ToString(dataRow[5]));
        customerObject[i].setTown(Convert.ToString(dataRow[6]));
        customerObject[i].setPostcode(Convert.ToString(dataRow[7]));
        customerObject[i].setEmail(Convert.ToString(dataRow[8]));
        customerObject[i].setPhone(Convert.ToString(dataRow[9]));
    }
}
```

Return to the **CustomerDetails** form and add a **loadCustomers()** method. This will call the method in the **customer** class which loads records from the database and creates a set of **customerObjects**. A loop then accesses each object and adds the customer name to the comboBox list.

Call **loadCustomers()** from the **CustomerDetails()** method.

```
public CustomerDetails()
{
    InitializeComponent();
    loadCustomers();
}

private void loadCustomers()
{
    customer.loadCustomers();
    combCustomers.Items.Clear();
    for (int i = 0; i < customer.customerCount; i++)
    {
        string customerName = customer.customerObject[i].getSurname()
            + ", " + customer.customerObject[i].getTitle() + " "
            + customer.customerObject[i].getForename();
        combCustomers.Items.Add(customerName);
    }
}
```

Run the program, move through to the **CustomerDetails** page and check that the comboBox displays a list of your customer test data.

The next step is to transfer data to the textBoxes when an existing customer is selected. Double click the customers **comboBox** and add code to the method.

```
private void combCustomers_SelectedIndexChanged(object sender, EventArgs e)
{
    int i = combCustomers.SelectedIndex;
    combTitle.Text = customer.customerObject[i].getTitle();
    txtForename.Text = customer.customerObject[i].getForename();
    txtSurname.Text = customer.customerObject[i].getSurname();
    txtAddress1.Text = customer.customerObject[i].getAddress1();
    txtAddress2.Text = customer.customerObject[i].getAddress2();
    txtTown.Text = customer.customerObject[i].getTown();
    txtPostcode.Text = customer.customerObject[i].getPostcode();
    txtEmail.Text = customer.customerObject[i].getEmail();
    txtPhone.Text = customer.customerObject[i].getPhone();
}
```

Run the program and check that customer contact information is displayed correctly when a customer name is selected from the comboBox.

Care must be taken to avoid existing customer records being saved for a second time, producing duplicate records in the database. Only records for new customers should be saved. We can control this by means of a boolean '**True / False**' variable to indicate whether we have entered a new customer.

Add the boolean variable at the start of **CustomerDetails**.

```
public partial class CustomerDetails : Form
{
    int f_performanceID;
    double f_totalCost;
    int[,] f_seatStatus;
    string f_eventTitle;

    bool newCustomer = true;

    public CustomerDetails()
    {
        InitializeComponent();

        loadCustomers();
    }
}
```

We will begin by assuming that a new customer is being entered. If, however, an existing customer is selected from the comboBox list, then **newCustomer** will be set to '**false**'. Add a line to the start of the comboBox_Click method to do this.

```
private void combCustomers_SelectedIndexChanged(object sender, EventArgs e)
{
    newCustomer = false;

    int i = combCustomers.SelectedIndex;
    combTitle.Text = customer.customerObject[i].getTitle();
}
```

Modify the '**continue to payment**' button_Click method so that records are only saved for new customers.

```
private void btnContinue_Click(object sender, EventArgs e)
{
    customerID = 0;

    if (newCustomer == true)
    {
        if (txtSurname.Text.Length > 0 && txtForename.Text.Length > 0 &&
            txtAddress1.Text.Length > 0 && txtPostcode.Text.Length > 0)
        {
            customerID = customer.saveCustomer(txtForename.Text, txtSurname.Text,
                combTitle.Text, txtAddress1.Text, txtAddress2.Text, txtTown.Text,
                txtPostcode.Text, txtEmail.Text, txtPhone.Text);
        }
        else
        {
            MessageBox.Show(
                "Missing customer information - name, address, postcode are required");
        }
    }
    else
    {
        int i = combCustomers.SelectedIndex;
        customerID = customer.customerObject[i].getCustomerID();
    }
}
```

Run the program. Select an exiting customer from the comboBox list, then click the '**continue to payment**' button. Exit from the program, then connect to the database. Check that a duplicate copy of the customer record has NOT been saved into the database table. Delete the connection to the database.

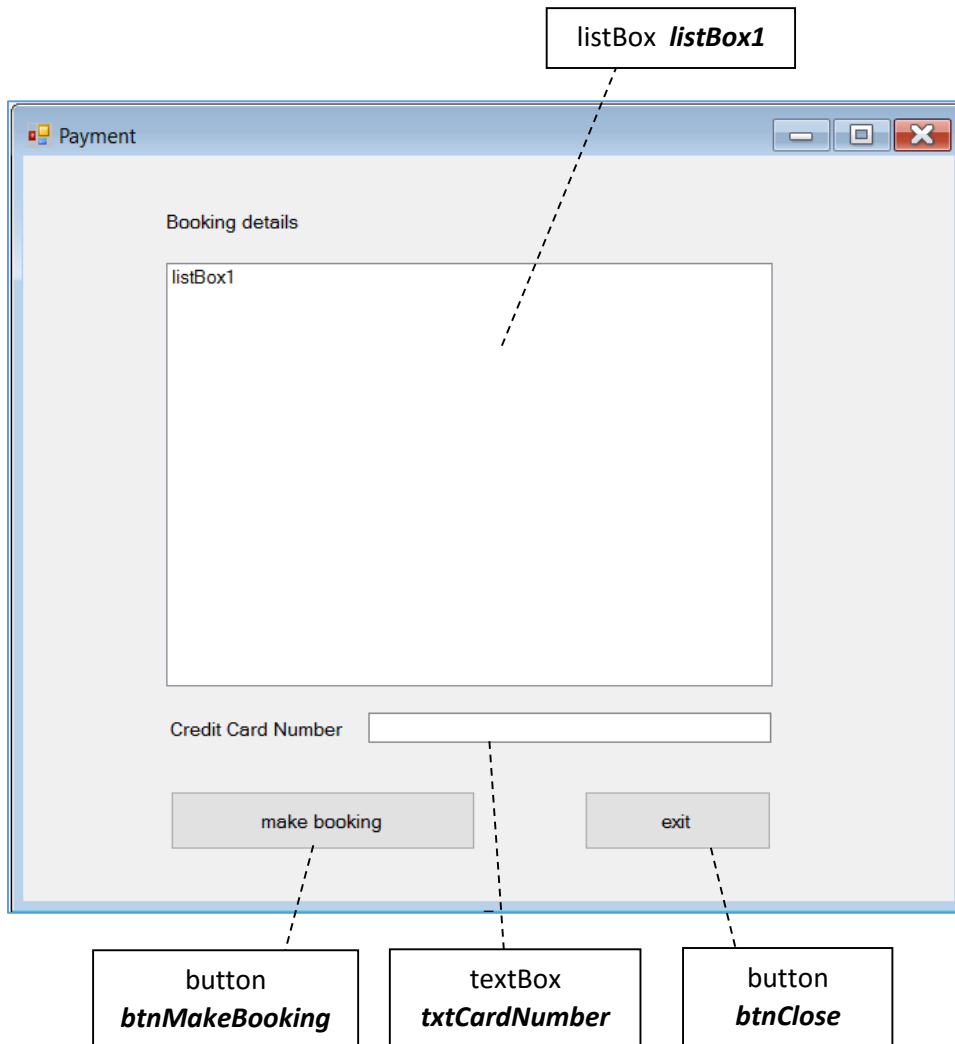
One final function to add to the **CustomerDetails** form is to clear all data from the textBoxes when the '**clear**' button is clicked.

Double click the '**clear**' button and add code to the button_Click method. Notice that the **newCustomer** variable is reset to '**true**', as the user may now want to enter the details of a new customer, rather than select a customer from the comboBox list.

```
private void btnClear_Click(object sender, EventArgs e)
{
    combCustomers.Text = "";
    combTitle.Text = "";
    txtForename.Clear();
    txtSurname.Clear();
    txtAddress1.Clear();
    txtAddress2.Clear();
    txtTown.Clear();
    txtPostcode.Clear();
    txtEmail.Clear();
    txtPhone.Clear();
    newCustomer = true;
}
```

This completes the collection of customer contact information, and the program can now proceed to the payment form.

Create a new **Windows Form** with the name '**Payment**', and add components as shown:



Add a line of code to the '**exit**' button_Click method.

```
private void btnClose_Click(object sender, EventArgs e)
{
    this.Close();
}
```

We will again transfer data from the **CustomerDetails** form, ready for use in saving the booking into the database. Add a set of variables at the start of the **Payment** form.

```
public partial class Payment : Form
{
    int f_performanceID;
    int f_customerID;
    double f_totalCost;
    int[,] f_seatStatus;
```


Begin a **getBooking()** method which will import data from the **CustomerDetails** form.

```
public void getBooking(int performanceID, double totalCost,
    int[,] seatStatus, int customerID, string eventTitle)
{
    f_performanceID = performanceID;
    f_customerID = customerID;
    f_totalCost = totalCost;
    f_seatStatus = seatStatus;
}
```

Return to the **CustomerDetails** form and add a block of code to the end of the '**continue to payment**' button_Click method. This will only operate if a customer record has been loaded from disc or a new customer record entered in the database, so that a valid **customerID** has been generated.

Booking data is transferred using the **getBooking()** method of the **Payment** form.

```
else
{
    int i = combCustomers.SelectedIndex;
    customerID = customer.customerObject[i].getCustomerID();
}

if (customerID > 0)
{
    Payment frmPayment = new Payment();
    frmPayment.getBooking(f_performanceID, f_totalCost,
        f_seatStatus, customerID, f_eventTitle);
    frmPayment.ShowDialog();
    this.Close();
}
```

Run the program and work through to the **CustomerDetails** form. Click the '**continue to payment**' button and check that the **Payment** form opens correctly.

The next task is to display details of the booking in the **listBox** on the **Payment** form, so that this can be confirmed as correct by the user before the booking is saved to the database.

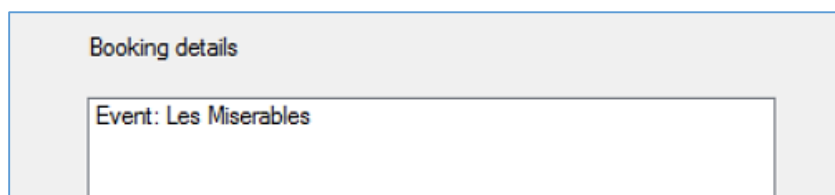
Go to the **getBooking()** method of the **Payment** form and add lines of code as shown below to clear the listBox and then display the title of the selected event.

```

public void getBooking(int performanceID, double totalCost, int[,]  
    seatStatus, int customerID, string eventTitle)  
{  
    f_performanceID = performanceID;  
    f_customerID = customerID;  
    f_totalCost = totalCost;  
    f_seatStatus = seatStatus;  
  
    listBox1.Items.Clear();  
    listBox1.Items.Add("Event: "+eventTitle);  
    listBox1.Items.Add("");  
}

```

Test the program to make sure the title of the chosen event is displayed:



We will next display details of the chosen performance. To do this, another method will be required in the **performance class** file which will load the details of a performance from the database using the **performanceID** value, then create a **performance** object from this data.

Go to the performance class file and add a **performanceDetails()** method.

(This is very similar to the **loadPerformances()** method, so you may save time by copying **loadPerformances()** and just making changes where necessary.)

```

public static void performanceDetails(int p)  
{  
    DataSet dsPerformances = new DataSet();  
  
    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;  
        AttachDbFilename=" + databaseLocation + "Integrated Security=True;  
        Connect Timeout=30; User Instance=True");  
    con.Open();  
    SqlCommand cmPerformances = new SqlCommand();  
    cmPerformances.Connection = con;  
    cmPerformances.CommandType = CommandType.Text;  
    cmPerformances.CommandText = "SELECT * FROM Performance"  
        + " WHERE performanceID='" + p + "'";  
    SqlDataAdapter daPerformances = new SqlDataAdapter(cmPerformances);  
    daPerformances.Fill(dsPerformances);  
    con.Close();  
  
    performanceObject[0] = new performance();  
    DataRow dataRow = dsPerformances.Tables[0].Rows[0];  
    performanceObject[0].setPerformanceID((int)dataRow[0]);  
    performanceObject[0].setEventID((int)dataRow[1]);  
    performanceObject[0].setDate(Convert.ToDateTime(dataRow[2]));  
    performanceObject[0].setTime(Convert.ToString(dataRow[3]));  
    performanceObject[0].setPrice(Convert.ToDouble(dataRow[4]));  
}  
}

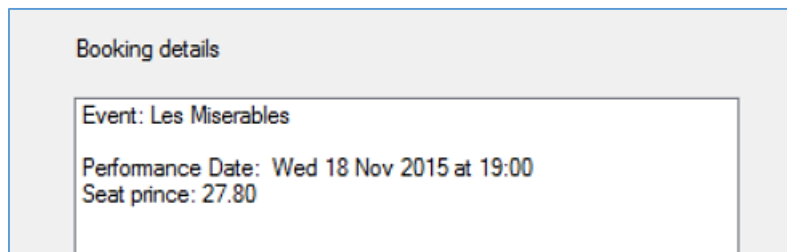
```

Return to the **Payment** form and add lines of code to the **getBooking()** method.

```
listBox1.Items.Clear();
listBox1.Items.Add("Event: "+eventTitle);
listBox1.Items.Add("");

performance.performanceDetails(performanceID);
DateTime performanceDate = performance.performanceObject[0].getDate();
string format = " ddd d MMM yyyy";
string performanceDateString = performanceDate.ToString(format);
string performanceTime = performance.performanceObject[0].getTime();
listBox1.Items.Add("Performance Date: "+performanceDateString + " at "
    + performanceTime);
string seatPrice=(performance.performanceObject[0].getPrice()).ToString("f2");
listBox1.Items.Add("Seat price: " + seatPrice);
```

Run the program and check that performance details are displayed correctly.



We will now list the rows and numbers of the selected seats. This can be done from data carried over in the **seatStatus** array. Selected seats will have a **seatStatus** value of 2. Add further lines of code to the **getBooking()** method.

```
listBox1.Items.Add("Performance Date: "+performanceDateString + " at "
    + performanceTime);
string seatPrice=(performance.performanceObject[0].getPrice()).ToString("f2");
listBox1.Items.Add("Seat price: " + seatPrice);

listBox1.Items.Add("");
for (int j = 1; j <= 11; j++)
{
    for (int i = 1; i <= 20; i++)
    {
        if (seatStatus[i, j] == 2)
        {
            char c = Convert.ToChar(64 + j);
            if (j > 8)
                c = Convert.ToChar(65 + j);
            listBox1.Items.Add("Row " + c + " Seat " + i);
        }
    }
}
listBox1.Items.Add("");
listBox1.Items.Add("Total cost £ " + totalCost.ToString("f2"));
listBox1.Items.Add(" ");
```

Run the program. Make a booking for several seats, and check that the row and seat numbers are displayed correctly. Check also that the total ticket cost is carried over correctly to the **Payment** form.

Booking details

Event: Les Miserables

Performance Date: Wed 18 Nov 2015 at 19:00

Seat price: 27.80

Row B Seat 8

Row B Seat 9

Row B Seat 10

Total cost £ 83.40

The final block of information to be displayed in the list box is the name and address of the customer. Another method will be required in the **customer** class file which will load the details for a customer using their **customerID** value, then create a **customer** object from this data.

Go to the **customer class** file and add a **customerDetails()** method. (This is very similar to the **loadCustomers()** method, so you may save time by copying and pasting.)

```
public static void customerDetails(int c)
{
    DataSet dsCustomers = new DataSet();
    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;
        AttachDbFilename=" + databaseLocation + "Integrated Security=True;
        Connect Timeout=30; User Instance=True");
    con.Open();
    SqlCommand cmCustomers = new SqlCommand();
    cmCustomers.Connection = con;
    cmCustomers.CommandType = CommandType.Text;
    cmCustomers.CommandText = "SELECT * FROM Customer"
        + " WHERE customerID ='" + c + "'";
    SqlDataAdapter daCustomers = new SqlDataAdapter(cmCustomers);
    daCustomers.Fill(dsCustomers);
    con.Close();

    customerObject[0] = new customer();
    DataRow dataRow = dsCustomers.Tables[0].Rows[0];
    customerObject[0].setCustomerID((int)dataRow[0]);
    customerObject[0].setForename(Convert.ToString(dataRow[1]));
    customerObject[0].setSurname(Convert.ToString(dataRow[2]));
    customerObject[0].setTitle(Convert.ToString(dataRow[3]));
    customerObject[0].setAddress1(Convert.ToString(dataRow[4]));
    customerObject[0].setAddress2(Convert.ToString(dataRow[5]));
    customerObject[0].setTown(Convert.ToString(dataRow[6]));
    customerObject[0].setPostcode(Convert.ToString(dataRow[7]));
    customerObject[0].setEmail(Convert.ToString(dataRow[8]));
    customerObject[0].setPhone(Convert.ToString(dataRow[9]));
}
```


Add the properties for the **booking** class, and the methods for transferring data into and out of the **booking objects**.

```
class booking
{
    private int bookingID;
    private int customerID;
    private int performanceID;
    private double totalCost;
    private string creditCardNumber;

    public void setBookingID(int b)
    {
        bookingID = b;
    }

    public int getBookingID()
    {
        return bookingID;
    }

    public void setCustomerID(int c)
    {
        customerID = c;
    }

    public int getCustomerID()
    {
        return customerID;
    }

    public void setPerformanceID(int p)
    {
        performanceID = p;
    }

    public int getPerformanceID()
    {
        return performanceID;
    }

    public void setTotalCost(double t)
    {
        totalCost = t;
    }

    public double getTotalCost()
    {
        return totalCost;
    }

    public void setCreditCardNumber(string cn)
    {
        creditCardNumber = cn;
    }

    public string getCreditCardNumber()
    {
        return creditCardNumber;
    }
}
```

When a booking is saved, the computer must carry out two tasks:

- Add a record to the **booking** table of the database. This will contain details of the event, performance, seats selected, ticket cost and identification of the customer.
- The **seat** records for the selected seats must be updated to show them as no longer available. The record for each booked seat should include the **bookingID** value, as a link to the details of the booking.

We will begin by writing an **AddBooking()** method for the **booking class**. It is also necessary to add '**using SqlClient**' and '**using Data**' directives, and to give the database location.

The **AddBooking()** method uses the **SELECT SCOPE_IDENTITY()** command to obtain the bookingID value allocated to the new record.

```
using System.Linq;
using System.Text;

using System.Data.SqlClient;
using System.Data;

namespace theatreBookings
{
    class booking
    {
        public static string databaseLocation="C:\\C#\\theatreBookings.mdf;";

        private int bookingID;
        private int customerID;
        private int performanceID;
        private double totalCost;
        private string creditCardNumber;

        public static int AddBooking(int performanceID, int customerID,
            double t, string cn, int[,] seatStatus)
        {
            SqlConnection con = new SqlConnection(@"Data Source=.\\SQLEXPRESS;
                AttachDbFilename="+databaseLocation + "Integrated Security=True;
                Connect Timeout=30; User Instance=True");
            con.Open();
            SqlCommand cmBooking = new SqlCommand();
            cmBooking.Connection = con;
            cmBooking.CommandType = CommandType.Text;
            cmBooking.CommandText="INSERT INTO Booking(customerID,performanceID,
                totalCost, creditCardNo)" + "VALUES ('" + customerID + "','" +
                performanceID + "','" + t + "','" + cn + "');"
            cmBooking.ExecuteNonQuery();
            cmBooking.CommandText = "SELECT SCOPE_IDENTITY()";
            int identity = Convert.ToInt32(cmBooking.ExecuteScalar());
            con.Close();
            return identity;
        }
    }
}
```

Return to the **Payment** form and double click the '**make booking**' button to create a `button_Click` method. Add code to call the **AddBooking()** method in the **booking** class, which will save the booking record in the database.

```
private void btnMakeBooking_Click(object sender, EventArgs e)
{
    string creditCardNumber = txtCardNumber.Text;

    try
    {
        int bookingID = booking.AddBooking(f_performanceID, f_customerID,
            f_totalCost, creditCardNumber, f_seatStatus);
    }
    catch
    {
        MessageBox.Show("ERROR");
    }
}
```

Run the program and make a booking, then click the '**make booking**' button. Keep a note of the event, performance and customer selected.

The screenshot shows a form with the following text:

Event: Les Miserables
 Performance Date: Wed 18 Nov 2015 at 19:00
 Seat price: 27.80
 Row C Seat 9
 Row C Seat 10
 Row C Seat 11
 Total cost £ 83.40
 Mrs Judith Williams
 36 Heol Glyndwr,
 Machynlleth, LL31 5UP

At the bottom, there is a label 'Credit Card Number' followed by a text box containing the value '4501348795678112'.

Exit from the program and go to the **Server Explorer** window. Connect the database and open the **Booking** table. Check that the booking has been saved correctly. You may need to use the **performanceID** and **customerID** values to look up details in the **Performance** and **Customer** tables.

	bookingID	customerID	performanceID	totalCost	creditCardNo
	1	4	9	83.40	4501348795678112
	NULL	NULL	NULL	NULL	NULL

After checking the tables, delete the connection to the database.

The final step in recording the booking is to update the seats as no longer available. We will need to add a method to the **seat** class to do this.

Open the seat class file and add an **updateSeat()** method.

```
public static int seatCount;
public static seat[] seatObject = new seat[240];

public static void updateSeat(int performanceID, char seatRow,
    int seatNumber, int bookingID)
{
    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;
        AttachDbFilename=" + databaseLocation + "Integrated Security=True;
        Connect Timeout=30; User Instance=True");
    con.Open();
    SqlCommand cmSeats = new SqlCommand();
    cmSeats.Connection = con;
    cmSeats.CommandType = CommandType.Text;
    cmSeats.CommandText = "UPDATE Seat SET available='1', bookingID='"
        + bookingID + "' WHERE performanceID='" + performanceID
        + "' AND seatRow='" + seatRow + "' AND seatNumber='" + seatNumber + "'";
    cmSeats.ExecuteNonQuery();
    con.Close();
}
```

Return to the **Payment** form and modify the **'make booking'** button_Click method to allow seats to be updated. Begin by adding variables to identify a seat, and a message box to confirm to the user that the booking has been saved successfully. The **Payment** form can then be closed, so that the user returns to the **DisplayEvents** page at the end of the booking procedure.

```
private void btnMakeBooking_Click(object sender, EventArgs e)
{
    string creditCardNumber = txtCardNumber.Text;

    char seatRow;
    int seatNumber;
    int rowNumber;

    try
    {
        int bookingID = booking.AddBooking(f_performanceID, f_customerID,
            f_totalCost, creditCardNumber, f_seatStatus);

        MessageBox.Show("Booking completed");
        this.Close();
    }
    catch
    {
        MessageBox.Show("ERROR");
    }
}
```

We can now add the code which updates the seat records. Seat data is first loaded, then a loop accesses each object in the **seat** array. If this seat has a **seatStatus** value of 2, indicating that it has been selected for the booking, then it will be marked as '**booked**' by calling the **updateSeat()** method in the **seat** class.

```
try
{
    int bookingID = booking.AddBooking(f_performanceID, f_customerID,
        f_totalCost, creditCardNumber, f_seatStatus);

    seat.loadSeats(f_performanceID);
    for (int i = 0; i < seat.seatCount; i++)
    {
        seatRow = seat.seatObject[i].getSeatRow();
        seatNumber = seat.seatObject[i].getSeatNumber();

        rowNumber = (int)seatRow - 64;
        if (rowNumber > 8)
            rowNumber--;

        if (f_seatStatus[seatNumber, rowNumber] == 2)
        {
            seat.updateSeat(f_performanceID, seatRow, seatNumber, bookingID);
        }
    }

    MessageBox.Show("Booking completed");
    this.Close();
}
```

Run the program and make a booking for several seats. Keep a record of the performance and seats selected.

The screenshot displays the 'Othello' application window. At the top, there is a 'Performance' dropdown menu set to 'Sat 10 Oct 2015 at 19:00'. Below this is a large grid of seat icons arranged in a trapezoidal shape, representing rows A through L and seats 1 through 16. Row A is at the top, and Row L is at the bottom. Seats are represented by small square icons. Some seats are highlighted in red (e.g., Row A, Seats 4, 5, 6, 7), and two seats in Row C (Seats 8 and 9) are highlighted in green and circled with a blue border, indicating they are selected. To the right of the grid, the row labels A through L are listed. At the bottom of the window, there is a 'Seats selected' section with a text box containing 'Row C Seat 8' and 'Row C Seat 9'. To the right of this, the 'Seat price' is listed as '£ 21.50', and the 'Total cost' is listed as '£ 43.00'.

Complete the booking

Booking details

Event: Othello

Performance Date: Sat 10 Oct 2015 at 19:00

Seat price: 21.50

Row C Seat 8

Row C Seat 9

Total cost £ 43.00

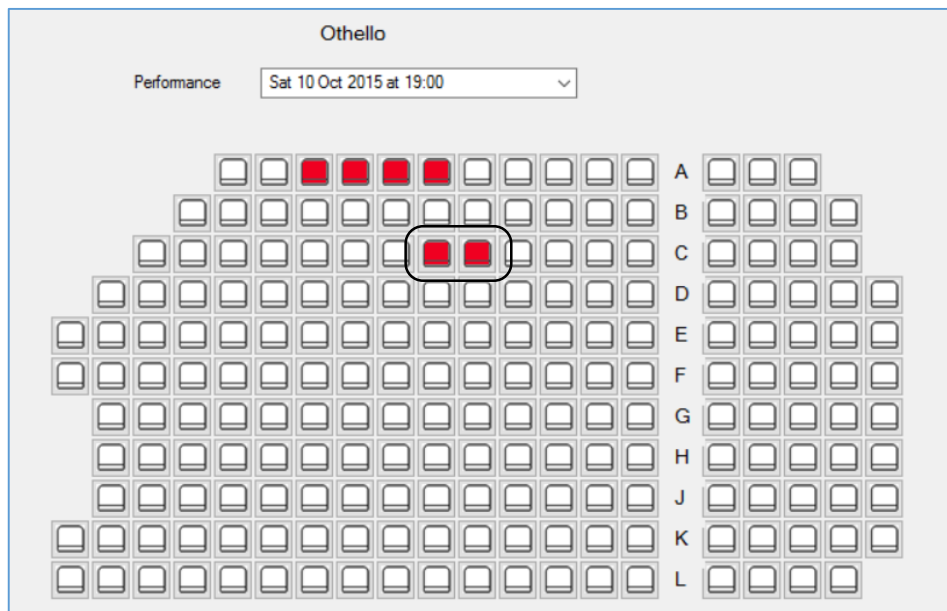
Ms Ruth Southhill
17 West Beach,
Fairbourn, LL45 6TW

Credit Card Number 5978456123456112

Booking completed

OK

Return to the same performance and confirm that the seats selected previously are now displayed as **'booked'**.



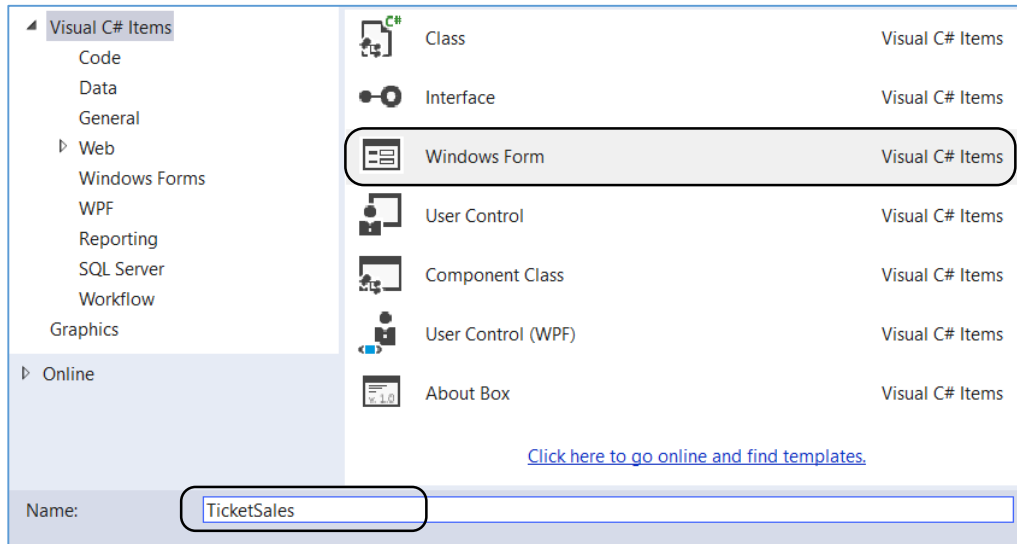
As a final check that the system is working correctly, exit from the program and go to the Server Explorer. Connect the database and confirm that the correct **bookingID** is shown alongside the booked seats.

seatRow	seatNumber	performanceID	available	bookingID
C	7	2	0	0
C	8	2	1	2
C	9	2	1	2
C	10	2	0	0
C	11	2	0	0

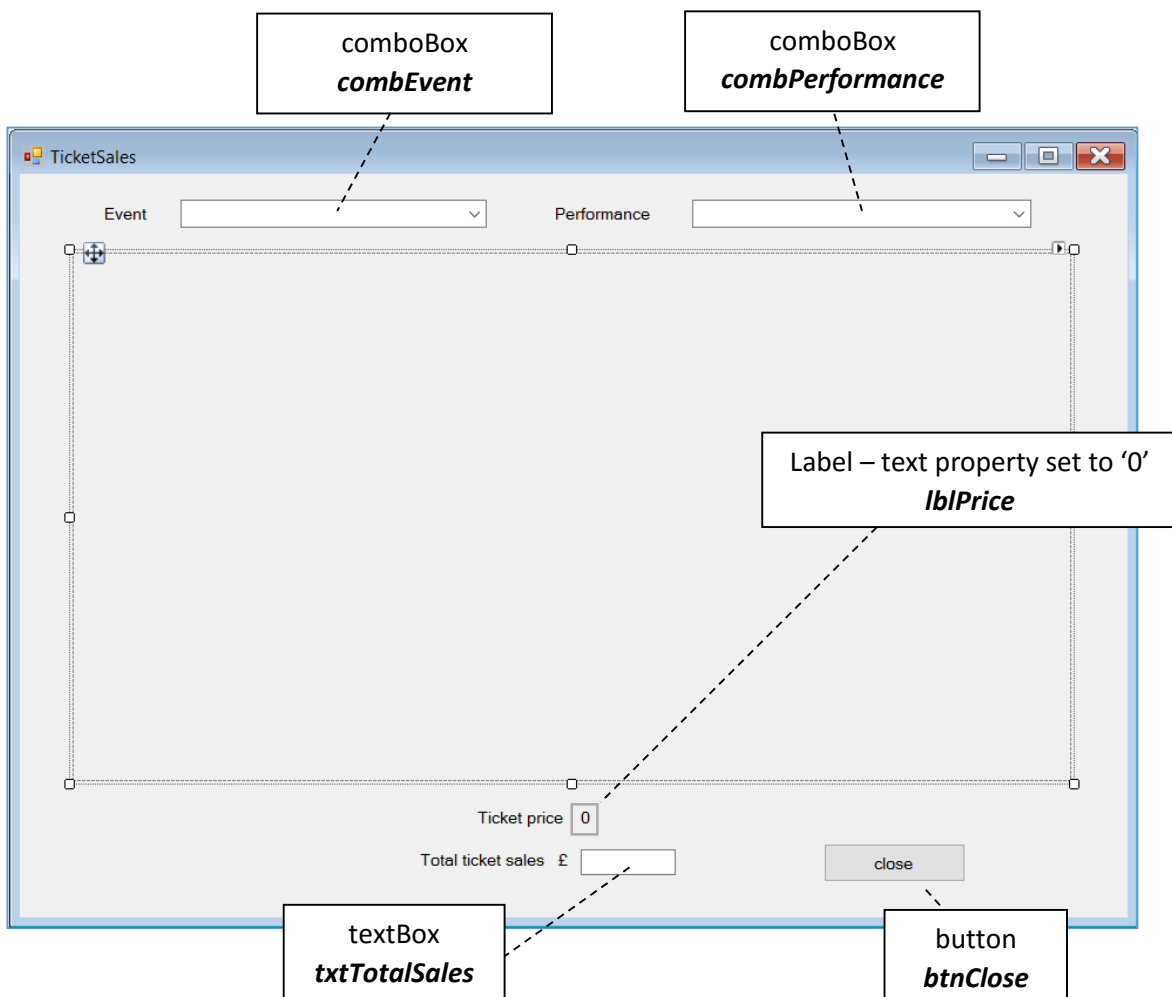
This completes the booking procedure.

We have two further sections of the program to develop. These will provide information for the theatre staff about bookings received and customer contact details.

Go to the Solution Explorer window. Right click the ***theatreBookings*** program icon to add a new ***Windows Form***. Give this the name ***TicketSales***.



Add components to the form.



Open the code window for the ***TicketSales*** form and add lines to the ***TicketSales()*** method to load the titles of theatre events into the comboBox.

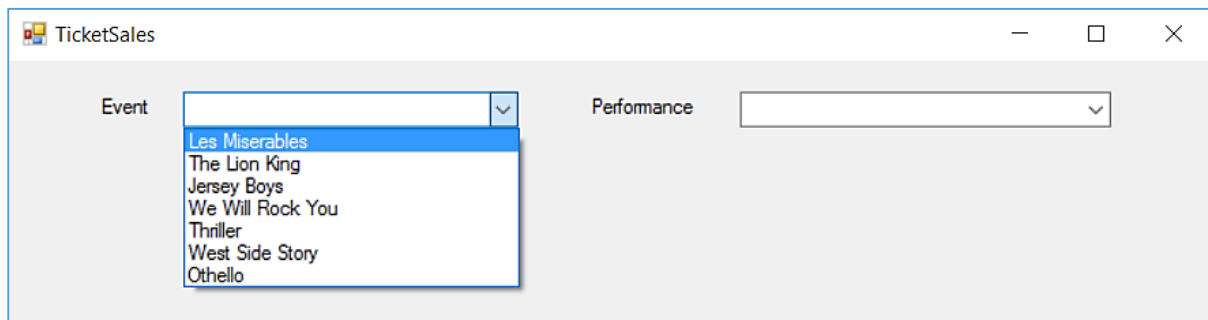
```
public TicketSales()
{
    InitializeComponent();

    combEvent.Items.Clear();
    string eventTitle;
    for (int i = 0; i < theatreEvent.eventCount; i++)
    {
        eventTitle = theatreEvent.eventObject[i].getTitle();
        combEvent.Items.Add(eventTitle);
    }
}
```

Go to the ***DisplayEvents*** form and double click the '***Display ticket sales for performance***' menu option. Add code to the menu_Click method to open the ***TicketSales*** form.

```
private void displayTicketSalesForPerformanceToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    TicketSales frmTicketSales = new TicketSales();
    frmTicketSales.ShowDialog();
}
```

Run the program and check that a list of event titles is displayed.



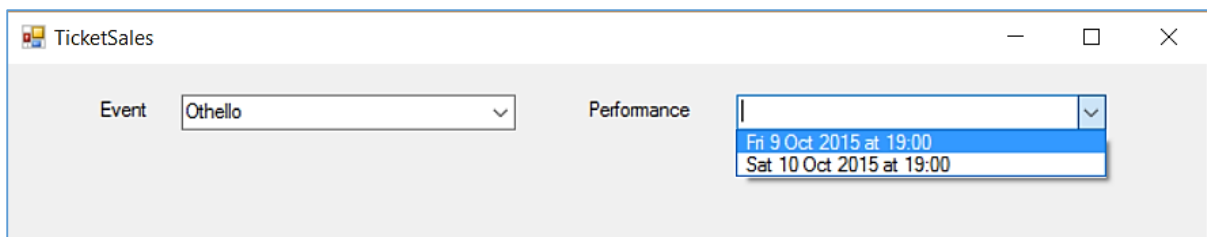
Return to the ***TicketSales*** form and add code to the '***close***' button.

```
private void btnClose_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Double click the **event comboBox** to create a method. Add code which will load performance dates and times for the selected event.

```
private void combEvent_SelectedIndexChanged(object sender, EventArgs e)
{
    int i = combEvent.SelectedIndex;
    int eventID = theatreEvent.eventObject[i].getEventID();
    performance.loadPerformances(eventID);
    combPerformance.Items.Clear();
    for (int p = 0; p < performance.performanceCount; p++)
    {
        DateTime performanceDate = performance.performanceObject[p].getDate();
        string format = "ddd d MMM yyyy";
        string performanceDateString = performanceDate.ToString(format);
        string performanceTime = performance.performanceObject[p].getTime();
        combPerformance.Items.Add(performanceDateString + " at " +
            performanceTime);
    }
}
```

Run the program to check that performance details are displayed correctly.



We can make use of the **drawPlan()** method which we wrote earlier for the **TheatrePlan** form.

Begin by adding the **button**, **label** and **seatStatus** arrays at the start of the **TicketSales** form.

```
public partial class TicketSales : Form
{
    Button[,] btnSeat = new Button[22, 12];
    Label[] label = new Label[12];
    int[,] seatStatus = new int[21, 12];
}
```

Copy the **drawPlan()** method from **TheatrePlan** into the **TicketSales** form.

The seat plan will be for display only, and it is not necessary for the seat buttons to be interactive. Remove the lines of code which allocate names to the seat buttons and create **button_Click** methods, as indicated on the next page.

```

private void drawPlan()
{
    int offset;
    int seatMax;
    for (int j = 1; j <= 11; j++)
    {
        seatMax = 20;
        if (j == 1) seatMax = 14;
        if (j == 2) seatMax = 16;
        if (j == 3) seatMax = 17;
        if (j == 4) seatMax = 19;
        if (j >= 7 && j <= 9) seatMax = 19;
        if (j == 11) seatMax = 19;
        for (int i = 1; i <= seatMax; i++)
        {
            offset = 0;
            if (j == 1) offset = 4;
            if (j == 2) offset = 3;
            if (j == 3) offset = 2;
            if (j == 4) offset = 1;
            if (j >= 7 && j <= 9) offset = 1;
            btnSeat[i, j] = new Button();
            btnSeat[i, j].Width = 28;
            btnSeat[i, j].Height = 28;
            btnSeat[i, j].Left = ((28 * i) + (28 * offset));
            if ((i + offset) > 15)
                btnSeat[i, j].Left += 28;
            btnSeat[i, j].Top = (28 * j);
            btnSeat[i, j].Image = Image.FromFile(".././seat1.png");
            if (seatStatus[i, j] == 1)
                btnSeat[i, j].Image = Image.FromFile(".././seat2.png");

            String buttonName = "btn";
            if (j <= 9)
                buttonName += " ";
            buttonName += j;
            if (i <= 9)
                buttonName += " ";
            buttonName += i;
            btnSeat[i, j].Name = buttonName;
            btnSeat[i, j].Click += new EventHandler(seat_Click);

            pnlTheatre.Controls.Add(btnSeat[i, j]);
            int rowNumber = j;
            if (j > 8)
                rowNumber++;
            string tooltipText = Convert.ToChar(rowNumber + 64).ToString()
                + (i).ToString();
            ToolTip buttonToolTip = new ToolTip();
            buttonToolTip.SetToolTip(btnSeat[i, j], tooltipText);
        }
        label[j] = new Label();
        label[j].Size = new System.Drawing.Size(24, 30);
        char c = Convert.ToChar(64 + j);
        if (j > 8)
            c = Convert.ToChar(65 + j);
        label[j].Text = Convert.ToString(c);
        label[j].Font = new System.Drawing.Font("Microsoft Sans Serif", 10F,
            System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point,
            ((byte)0));
        label[j].Location = new System.Drawing.Point(455, 4 + 28 * j);
        pnlTheatre.Controls.Add(label[j]);
    }
}

```

Remove these
lines of code

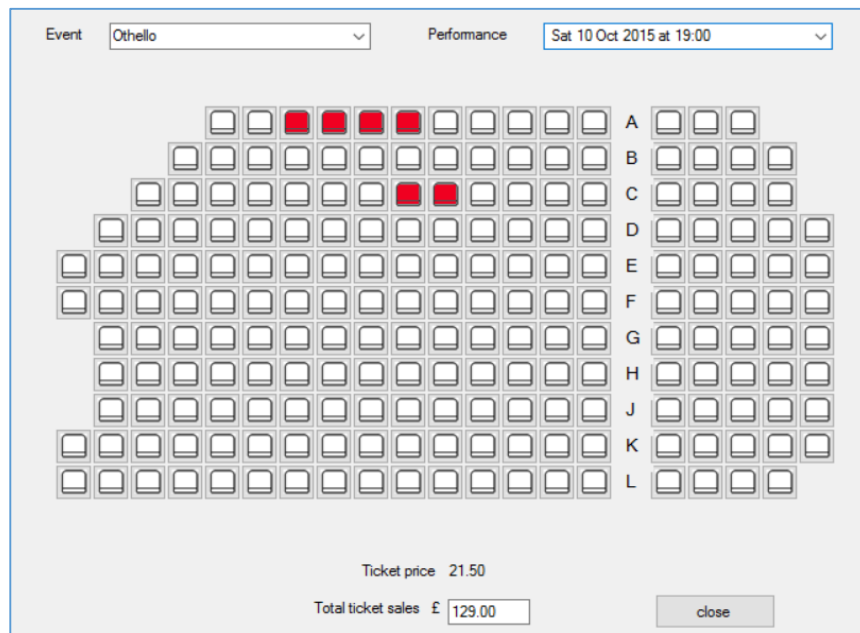
Double click the **performance comboBox** and add code to load the seat data for the selected performance. This data is transferred to the **seatStatus array**, using code values of 0 for an available seat and 1 for a booked seat. At the same time, the total cost of tickets for all booked seats is being calculated.

When seat data has been processed, the previous pattern of seat buttons are removed from the form, and the **drawPlan()** method is called to create the seating plan display.

```
private void combPerformance_SelectedIndexChanged(object sender, EventArgs e)
{
    int p = combPerformance.SelectedIndex;
    int performanceID = performance.performanceObject[p].getPerformanceID();
    double seatCost = performance.performanceObject[p].getPrice();

    seat.loadSeats(performanceID);
    double totalSales = 0;
    for (int i = 0; i < seat.seatCount; i++)
    {
        char c = seat.seatObject[i].getSeatRow();
        int r = ((int)c) - 64;
        if (r > 8)
            r--;
        int s = seat.seatObject[i].getSeatNumber();
        int available = seat.seatObject[i].getAvailable();
        seatStatus[s, r] = available;
        if (available == 1)
        {
            totalSales += seatCost;
        }
    }
    int totalButtons = pnlTheatre.Controls.Count;
    for (int i = 0; i < totalButtons; i++)
    {
        pnlTheatre.Controls.RemoveAt(0);
    }
    drawPlan();
    lblPrice.Text = seatCost.ToString("f2");
    txtTotalSales.Text = totalSales.ToString("f2");
}
```

Run the program and check that seat plans can be displayed for any performance.



The final option we will include in our program is to view customer records.

Add another Windows Form with the name '**CustomerRecord**'. Add components to the form as shown.

The screenshot shows a Windows Form titled 'CustomerRecord'. It contains the following components and labels:

- comboBox** *combCustomers*: A dropdown menu at the top.
- textBoxes**: A list of text boxes for form fields:
 - txtTitle*
 - txtForename*
 - txtSurname*
 - txtAddress1*
 - txtAddress2*
 - txtTown*
 - txtPostcode*
 - txtEmail*
 - txtPhone*
- listBox** *listBox1*: A list box labeled 'Bookings'.
- button** *btnClose*: A button labeled 'close'.

Labels are connected to their respective components by dashed lines.

Double click the '**close**' button and add a line of code.

```
private void btnClose_Click(object sender, EventArgs e)
{
    this.Close();
}
```

Create a **loadCustomers()** method which will load customer names into the drop-down comboBox list. Call this method from the **CustomerRecord()** method.

```
public CustomerRecord()
{
    InitializeComponent();
    loadCustomers();
}

private void loadCustomers()
{
    customer.loadCustomers();
    combCustomers.Items.Clear();
    for (int i = 0; i < customer.customerCount; i++)
    {
        string customerName = customer.customerObject[i].getSurname() + ", "
            + customer.customerObject[i].getTitle() + " "
            + customer.customerObject[i].getForename();
        combCustomers.Items.Add(customerName);
    }
}
```

Go to the **DisplayEvents** form and add code to the '**View customer records**' menu option to load the **CustomerRecord** form.

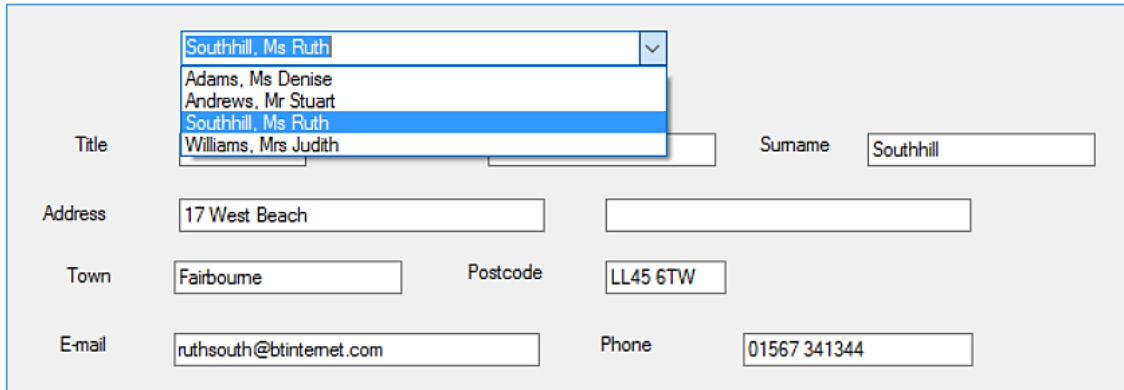
Run the program and check that customer names are listed correctly in the comboBox on the **CustomerRecord** form.

The screenshot shows the CustomerRecord form with a dropdown menu open, displaying a list of customer names: Adams, Ms Denise; Andrews, Mr Stuart; Southhill, Ms Ruth; and Williams, Mrs Judith. The form includes fields for Title, Surname, Address (split into two parts), Town, and Postcode.

Return to the CustomerRecord page and double click the **comboBox**. Add code to the method.

```
private void combCustomers_SelectedIndexChanged(object sender, EventArgs e)
{
    int i = combCustomers.SelectedIndex;
    int customerID=customer.customerObject[i].getCustomerID();
    txtTitle.Text = customer.customerObject[i].getTitle();
    txtForename.Text = customer.customerObject[i].getForename();
    txtSurname.Text = customer.customerObject[i].getSurname();
    txtAddress1.Text = customer.customerObject[i].getAddress1();
    txtAddress2.Text = customer.customerObject[i].getAddress2();
    txtTown.Text = customer.customerObject[i].getTown();
    txtPostcode.Text = customer.customerObject[i].getPostcode();
    txtEmail.Text = customer.customerObject[i].getEmail();
    txtPhone.Text = customer.customerObject[i].getPhone();
}
```

Run the program and select a customer from the comboBox on the **CustomerRecord** form. The customer name, address, e-mail and phone number should be displayed.



The screenshot shows a Windows form titled 'CustomerRecord'. It features a 'Title' label and a dropdown menu. The dropdown is open, displaying a list of customer names: 'Southhill, Ms Ruth' (highlighted), 'Adams, Ms Denise', 'Andrews, Mr Stuart', 'Southhill, Ms Ruth', and 'Williams, Mrs Judith'. Below the dropdown, the form fields are populated with the details of the selected customer: 'Address' is '17 West Beach', 'Town' is 'Fairbourne', 'E-mail' is 'ruthsouth@btinternet.com', 'Surname' is 'Southhill', 'Postcode' is 'LL45 6TW', and 'Phone' is '01567 341344'.

The final step is to display the details of the bookings made by the selected customer. Before doing this we must add a method to the booking class.

Open the **booking class file** and create a **loadBookings()** method. Also add a **bookingCount** variable, and an array to hold **bookingObjects**.

```
private double totalCost;
private string creditCardNumber;

public static int bookingCount;
public static booking[] bookingObject = new booking[16];

public static void loadBookings(int customerID)
{
    bookingCount = 0;
    DataSet dsBookings = new DataSet();
    SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;
        AttachDbFilename=" + databaseLocation + "Integrated Security=True;
        Connect Timeout=30; User Instance=True");
    con.Open();
    SqlCommand cmBookings = new SqlCommand();
    cmBookings.Connection = con;
    cmBookings.CommandType = CommandType.Text;
    cmBookings.CommandText = "SELECT * FROM Booking WHERE customerID = '"
        + customerID + "'";
    SqlDataAdapter daBookings = new SqlDataAdapter(cmBookings);
    daBookings.Fill(dsBookings);
    con.Close();
    bookingCount = dsBookings.Tables[0].Rows.Count;
    for (int i = 0; i < bookingCount; i++)
    {
        bookingObject[i] = new booking();
        DataRow dataRow = dsBookings.Tables[0].Rows[i];
        bookingObject[i].setBookingID((int)dataRow[0]);
        bookingObject[i].setPerformanceID((int)dataRow[2]);
        bookingObject[i].setTotalCost(Convert.ToDouble(dataRow[3]));
        bookingObject[i].setCreditCardNumber(Convert.ToString(dataRow[4]));
    }
}
```

Return to the **CustomerRecord** form and add code to the **Customers comboBox** `IndexChanged()` method. This loads the booking records and uses a loop to check if each booking has been made by the selected customer. If so, the details of the booked performance will be displayed in the `listBox` on the **CustomerRecord** form.

```
txtTown.Text = customer.customerObject[i].getTown();
txtPostcode.Text = customer.customerObject[i].getPostcode();
txtEmail.Text = customer.customerObject[i].getEmail();
txtPhone.Text = customer.customerObject[i].getPhone();

booking.loadBookings(customerID);
listBox1.Items.Clear();
for (int b = 0; b < booking.bookingCount; b++)
{
    int performanceID = booking.bookingObject[b].getPerformanceID();
    performance.performanceDetails(performanceID);
    DateTime performanceDate = performance.performanceObject[0].getDate();
    string format = "ddd d MMM yyyy";
    string performanceDateString = performanceDate.ToString(format);
    string performanceTime = performance.performanceObject[0].getTime();
    int eventID = performance.performanceObject[0].getEventID();
    for (int j = 0; j < theatreEvent.eventCount; j++)
    {
        int eventIDfound = theatreEvent.eventObject[j].getEventID();
        if (eventID == eventIDfound)
        {
            string eventTitle = theatreEvent.eventObject[j].getTitle();
            listBox1.Items.Add(eventTitle);
        }
    }
    listBox1.Items.Add("EventID: " + eventID);
    listBox1.Items.Add("");
    listBox1.Items.Add("PerformanceID: " + performanceID);
    listBox1.Items.Add(performanceDateString + " at " + performanceTime);
    listBox1.Items.Add("");
    int bookingID = booking.bookingObject[b].getBookingID();
    listBox1.Items.Add("BookingID: " + bookingID);
}
```

Run the program and check that the performances booked are displayed.

The screenshot shows a Windows application window titled 'CustomerRecord'. At the top, there is a dropdown menu showing 'Williams, Mrs Judith'. Below this are several input fields arranged in a form:

- Title: Mrs
- Forename: Judith
- Surname: Williams
- Address: 36 Heol Glyndwr
- Town: Machynlleth
- Postcode: LL31 5UP
- E-mail: williams56@gmail.com
- Phone: 01678 965477

 At the bottom, there is a section labeled 'Bookings' which contains a text box displaying the following information:

- Les Miserables
- EventID: 1
- PerformanceID: 9
- Wed 18 Nov 2015 at 19:00
- BookingID: 1

The only thing to now add is a list of the seats booked by the customer and payment details. Add code to the **Customers comboBox** IndexChanged() method to do this.

```

listBox1.Items.Add("PerformanceID: " + performanceID);
listBox1.Items.Add(performanceDateString + " at " + performanceTime);
listBox1.Items.Add("");
int bookingID = booking.bookingObject[b].getBookingID();
listBox1.Items.Add("BookingID: " + bookingID);

listBox1.Items.Add("Seats booked:");
seat.loadSeats(performanceID);
for (int k = 0; k < seat.seatCount; k++)
{
    int bookingIDfound = seat.seatObject[k].getBookingID();
    if (bookingID == bookingIDfound)
    {
        char seatRow = seat.seatObject[k].getSeatRow();
        string seatBooked = Convert.ToString(seatRow)
            + Convert.ToString(seat.seatObject[k].getSeatNumber());
        listBox1.Items.Add(seatBooked);
    }
}
listBox1.Items.Add("");
double totalCost = booking.bookingObject[b].getTotalCost();
listBox1.Items.Add("Total cost: £ " + totalCost.ToString("f2"));
listBox1.Items.Add("");
string creditCardNumber = booking.bookingObject[b].getCreditCardNumber();
listBox1.Items.Add("Payment by credit card " + creditCardNumber);
listBox1.Items.Add("_____");
listBox1.Items.Add("");
}

```

Run the program and check that seats booked are now listed.

The screenshot displays a web-based booking form for a theatre. At the top, a dropdown menu shows 'Williams, Mrs Judith'. Below this, the form is organized into several sections:

- Title:** A dropdown menu set to 'Mrs'.
- Forename:** A text input field containing 'Judith'.
- Surname:** A text input field containing 'Williams'.
- Address:** A text input field containing '36 Heol Glyndwr'.
- Town:** A text input field containing 'Machynlleth'.
- Postcode:** A text input field containing 'LL31 5UP'.
- E-mail:** A text input field containing 'williams56@gmail.com'.
- Phone:** A text input field containing '01678 965477'.
- Bookings:** A large text area displaying the following information:
 - Les Miserables
 - EventID: 1
 - PerformanceID: 9
 - Wed 18 Nov 2015 at 19:00
 - BookingID: 1
 - Seats booked:
 - Total cost: £ 83.40
 - Payment by credit card 4501348795678112

Congratulations on completing this final project successfully.