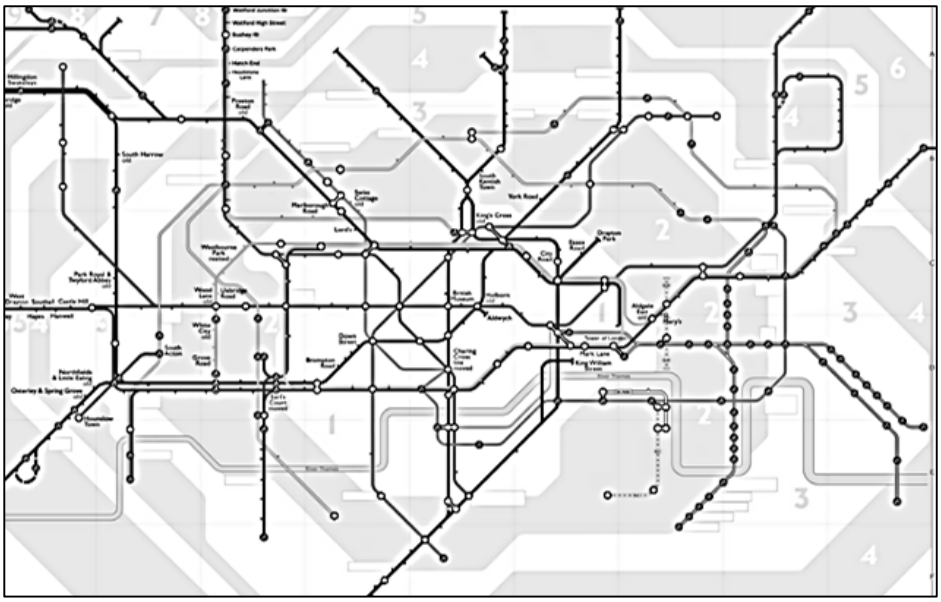
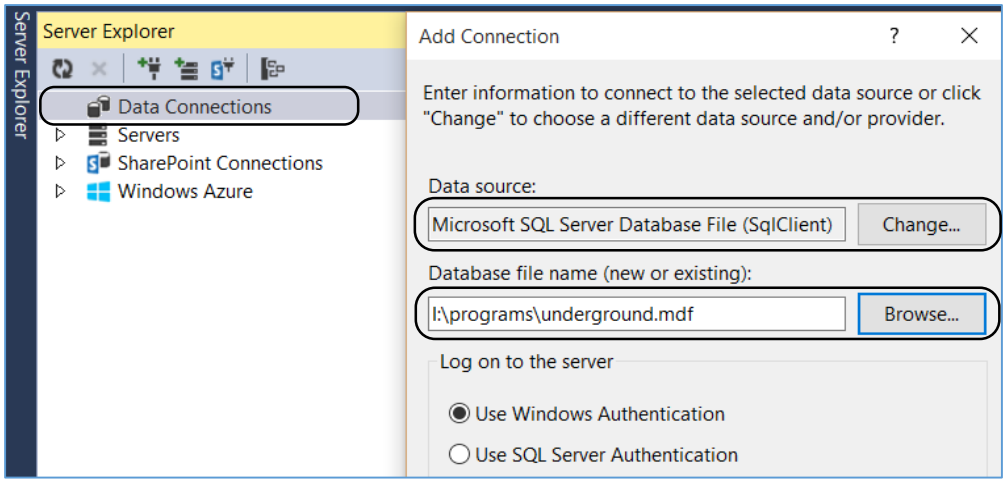
6 London Underground

In the next project we will investigate a more challenging program algorithm, to provide information to travellers about routes between stations on the London Underground system. This program will combine the use of database files and some new techniques for producing screen graphics.

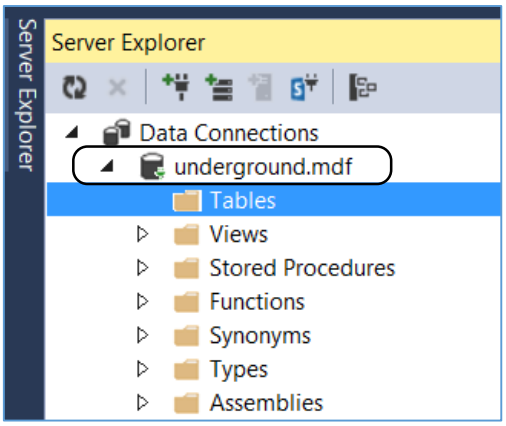


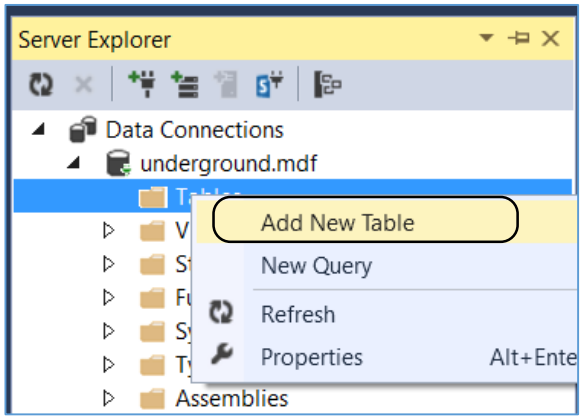
The London Underground is a large system of interconnected lines. To keep our project manageable, we will restrict the program to just four: the ***Circle***, ***Victoria***, ***Central*** and ***Northern*** lines, and we will include only a small sample of stations along these routes. The principles we develop, however, could be extended in a fairly straightforward way to cover the complete network.

We will begin by setting up a database to hold information about the stations and their map locations. Open Visual Studio, and display the ***Server Explorer*** window. Right click on Data Connections and select '***Add Connection***' . Set up a new ***Microsoft SQL Server Database File*** called '***underground***' in the same directory as your C# program folders.

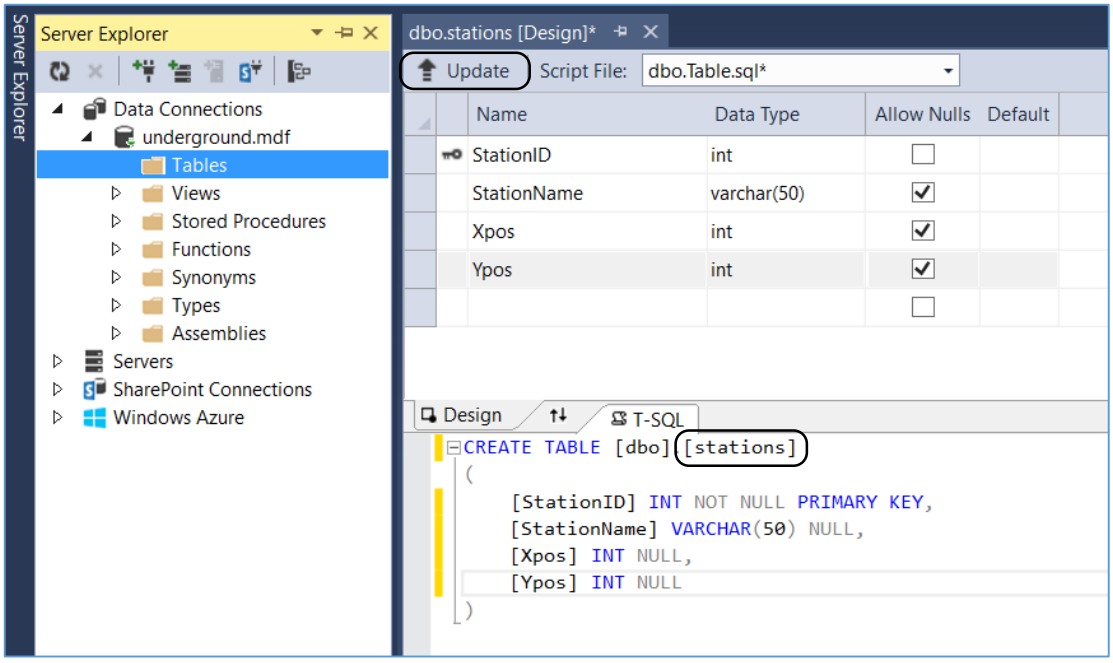


The new database '***underground***' should be listed under the ***Data Connections*** icon. Click the small arrow to the left to open the database. Right click on '***Tables***' and select '***Add New Table***'.





We are going to create a table which will store the names and map coordinates of a number of underground stations. Set up the Column Names and Data Types as shown:



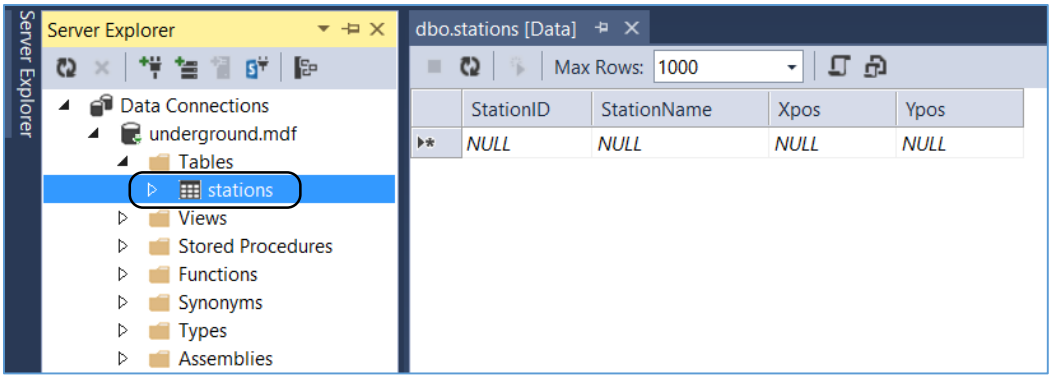
Go to the ***CREATE TABLE*** line of the SQL code and change the table name to '***stations***'.

Click the ***Update*** button. When the ***Database Updates*** window opens, click the '***Update Database***' button.

Finally, close the design window by clicking the small cross above the table.

Right-click on ***underground.mdf*** in the ***Server Explorer*** window and select '***Refresh***'.

Click the small arrow to the left of the ***Tables*** icon to show the '***stations***' table. Right click on '***stations***' and select '***Show Table Data***'. A blank table should appear:

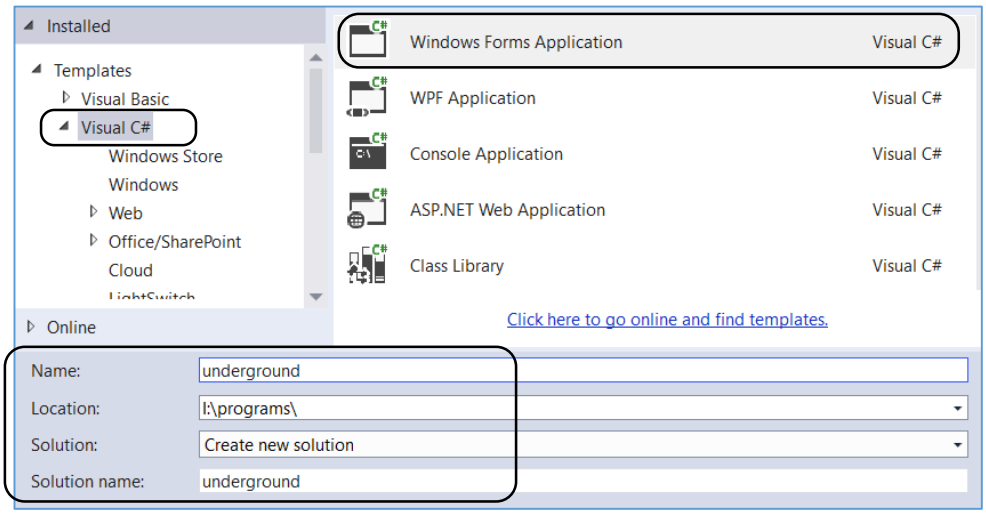


Enter the station information given in the list below. The map coordinates have been found for you by copying the London Underground map into the graphics utility program Paint, then reading off the coordinates as the cursor was moved over each of the required stations.

StationID	**StationName**	**Xpos** (pixels across)	**Ypos** (pixels down)
1	Ealing Broadway	105	269
2	Notting Hill Gate	270	268
3	Paddington	272	221
4	Euston	409	218
5	High Barnet	439	26
6	Oxford Circus	361	267
7	Tottenham Court Road	400	267
8	Embankment	400	333
9	Victoria	342	333
10	South Kensington	280	333
11	Stockwell	383	446
12	Morden	294	535
13	Brixton	410	472
14	Tower Hill	529	299
15	Liverpool Street	531	250
16	Epping	667	11
17	Walthamstow Central	629	119
18	Stratford	667	199
19	Kings Cross	440	218

Check your entries carefully, then close the table. Go to the ***Server Explorer*** window and delete the connection to the ***underground.mdf*** database.

We can now set up the C# program. Select '***FILE*** / ***New Project***'. Click '***Visual C#***' and '***Windows Forms Application***', and set the program name to '***underground***'.



Form1 will be created. Right click the form and select '***View Code***'.

The first stage is to load the station data from the database table.

- Add '***SqlClient***' to the list of '***using***' directives at the start of the program.
- Show the database location,
- Create a ***DataSet*** for the station data.
- Set up the ***GetStations( )*** method to load the station data from the database table. This will be very similar to the method you used to load flight data in the Airport program.
- Add a line to the ***Form1( )*** method to run ***GetStations( )*** when the program first starts.

```
using System.Windows.Forms;
using System.Data.SqlClient;

namespace underground
{
    public partial class Form1 : Form
    {
        string databaseLocation = "C:\\C#\\underground.mdf;";
        DataSet dsStations = new DataSet();

        public Form1()
        {
            InitializeComponent();

            GetStations();
        }

        private void GetStations()
        {

        }
    }
```

Add code to the ***GetStations( )*** method which will load the data and transfer it to the DataSet. Note that the line beginning:

SqlConnection con =

should be entered as a single line of code with no line breaks.

```
private void GetStations()
{
    try
    {
        SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;
           AttachDbFilename=" + databaseLocation + "Integrated Security=True;
           Connect Timeout=30; User Instance=True");
        con.Open();
        SqlCommand cmStations = new SqlCommand();
        cmStations.Connection = con;
        cmStations.CommandType = CommandType.Text;
        cmStations.CommandText = "SELECT * FROM stations ORDER BY stationName ASC";

        SqlDataAdapter daStations = new SqlDataAdapter(cmStations);
        daStations.Fill(dsStations);
        con.Close();
    }

    catch
    {
        MessageBox.Show("File error");
    }
}
```

We have added an instruction '***ORDER BY stationName ASC***' to the SQL command, which will sort the station records into alphabetical order of station name. This will be helpful later when the user is selecting their journey.

Once the data has been loaded, it will be more convenient to transfer it into arrays ready for processing. Set up four separate arrays for the stationID numbers, stationNames, and the X and Y map cooordinates. We will also set up an integer variable ***stationCount*** to record the number of stations for which we have data.

```
public partial class Form1 : Form
{
    string databaseLocation = "C:\\C#\\underground.mdf;";

    int stationCount;
    int[] stationID = new int[20];
    string[] stationName = new string[20];
    int[] stationX = new int[20];
    int[] stationY = new int[20];

    DataSet dsStations = new DataSet();

    public Form1()
    {
        InitializeComponent();
        GetStations();
    }
```

Add code to the ***GetStations( )*** method to determine the number of stations, and then use a loop to transfer the data for each station into the arrays:

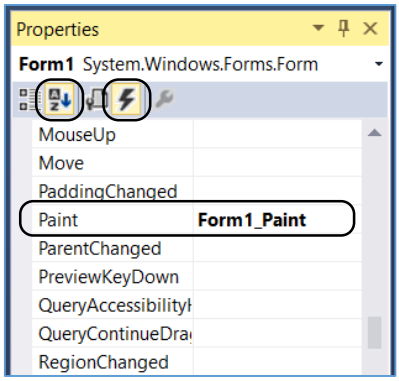
```
        catch
        {
            MessageBox.Show("File error");
        }
        stationCount = dsStations.Tables[0].Rows.Count;

        for (int i = 0; i < stationCount; i++)
        {
            DataRow drStation = dsStations.Tables[0].Rows[i];

            stationID[i] = Convert.ToInt16(drStation[0]);
            stationName[i] = Convert.ToString(drStation[1]);
            stationX[i] = Convert.ToInt16(drStation[2]);
            stationY[i] = Convert.ToInt16(drStation[3]);
        }
    }
```

The next step is to write a method ***DrawMap( )*** which will display the stations on screen as the basis of a route diagram for the railway system. Add this below the ***GetStations( )*** method:

```
        for (int i = 0; i < stationCount; i++)
        {
            DataRow drStation = dsStations.Tables[0].Rows[i];

            stationID[i] = Convert.ToInt16(drStation[0]);
            stationName[i] = Convert.ToString(drStation[1]);
            stationX[i] = Convert.ToInt16(drStation[2]);
            stationY[i] = Convert.ToInt16(drStation[3]);
        }
    }
    private void DrawMap()
    {
        Graphics g = this.CreateGraphics();
        Pen black = new Pen(Color.Black, 1);
        SolidBrush white = new SolidBrush(Color.White);
        Font font = new Font("FreightSans Medium", 7, FontStyle.Regular);

        for (int i = 0; i < stationCount; i++)
        {
            g.FillEllipse(white, stationX[i] - 5, stationY[i] - 5, 10, 10);
            g.DrawEllipse(black, stationX[i] - 5, stationY[i] - 5, 10, 10);
            g.DrawString(stationName[i], font, Brushes.Black,
                   new Rectangle(stationX[i] + 3, stationY[i] + 2, 61, 50));
        }
    }
```

This code sets up black line and white fill colours, then draws a circle in the correct X, Y map position for each station. The station name is then added as a caption.

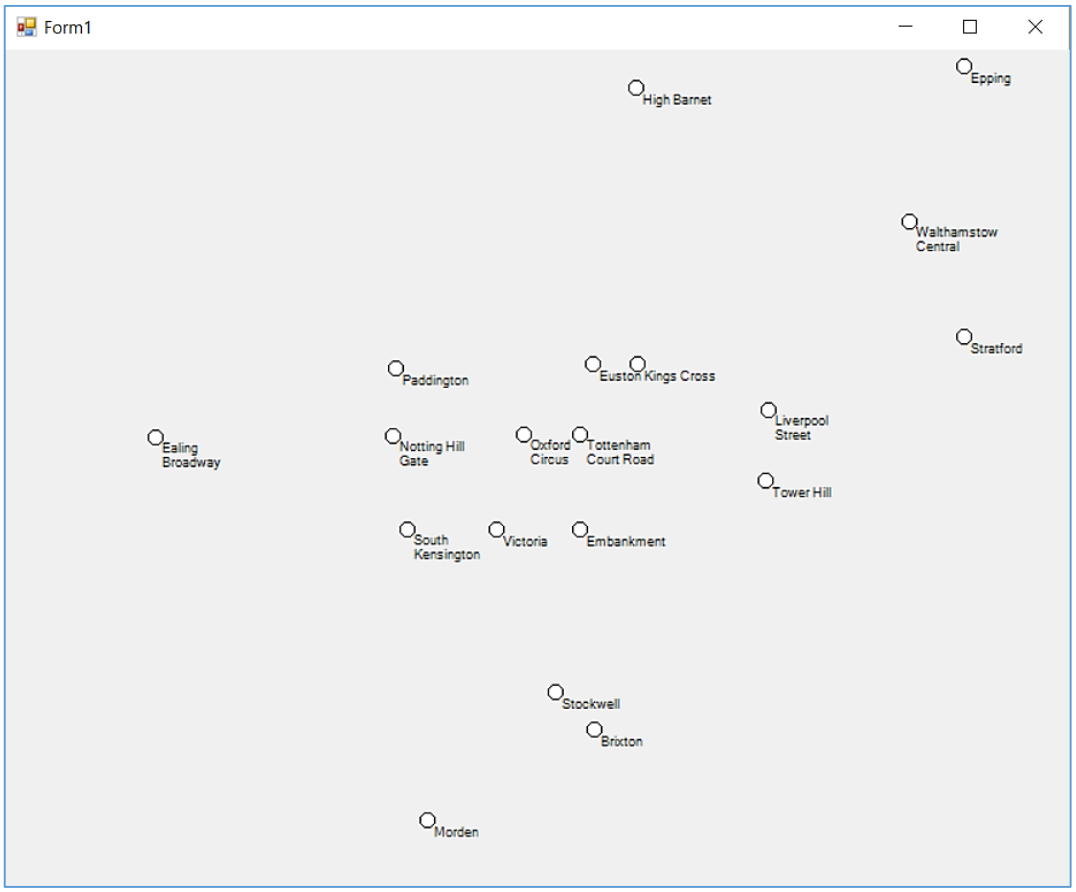
To make the graphics appear when the program runs, it is necessary to add a ***Paint( )*** method to the form. Change to the form ***design view***, click to select Form1, then go to the Properties window. Click the ***Events*** icon, checking that the ***alphabetical list*** icon is also selected:



Locate the '***Paint***' event and double click in the right column to create a ***Form1_Paint*** method. Add the ***DrawMap( )*** method to this:

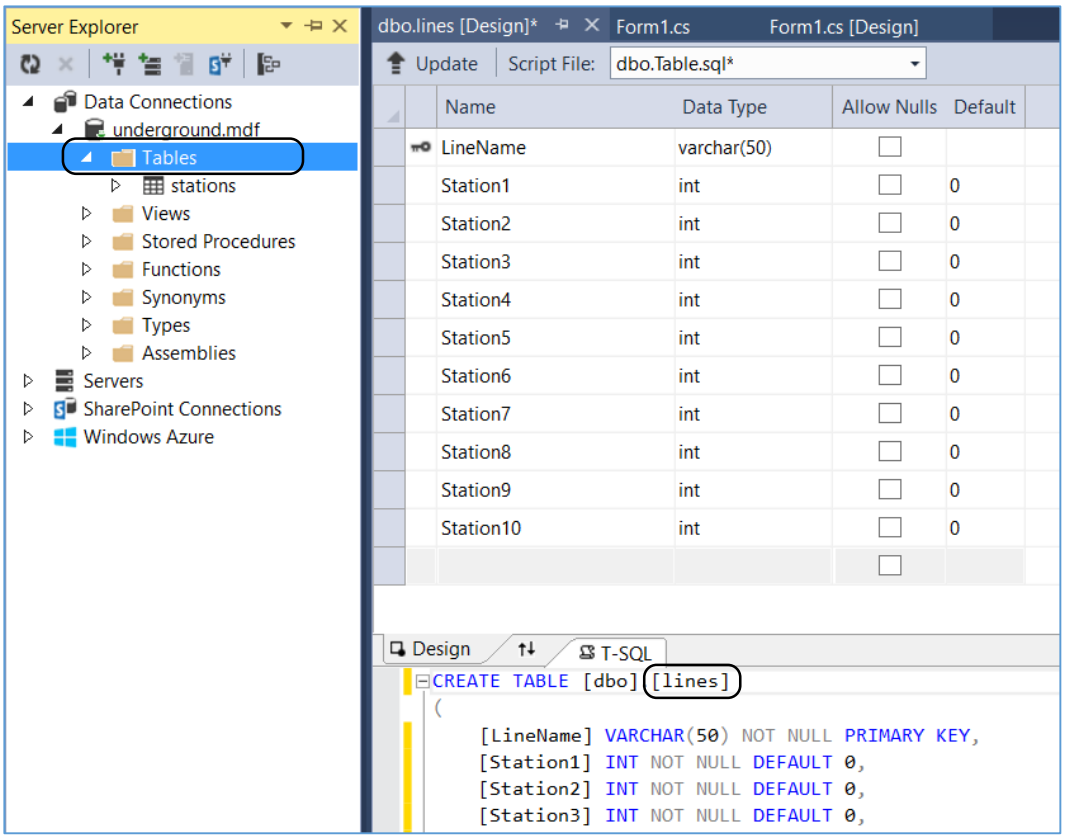
```
private void Form1_Paint(object sender, PaintEventArgs e)

{
    DrawMap();
}
```

Run the program. The stations should be displayed in the pattern shown. If the complete map area is not visible then close the program, return to the design view and enlarge the form. If any stations appear in an incorrect position, go to the database table and check that the X and Y cooordinates have been entered correctly.



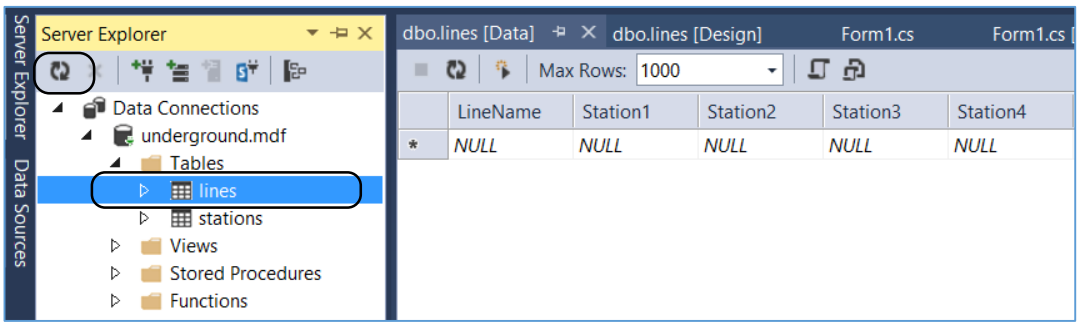
The next task is to connect the stations along each Underground Line. We will need another table in the database to record the sequence of stations along each line. Go to the ***Server Explorer*** window and right click the ***Data Connections*** icon. Re-connect the ***underground.mdf*** database.

Right-click the ***Tables*** icon. Select '***Add New Table***'. Set up the primary key field as '***LineName***', then add ten integer fields to represent the sequence of stations along the route. These fields are named '***Station1***' to '***Station10***'. For each of the station fields, remove the tick from ***Allow Nulls*** column and set a ***Default*** value of ***0***.



Change the table name to '***lines***'. Click the '***Update***' button and update the database.

Close the table design window. Click the ***refresh*** button in the top left corner of the Database Explorer window. The '***lines***' table should now appear in the list. Click right on the ***lines*** table icon and select '***Show Table Data***'. The empty table opens.



We are going to enter data for four Underground Lines. Each row begins with the name of the line, followed by the ID numbers of the stations along the line. These ID numbers were allocated by the computer when you entered the station data earlier.

LineName	Station1	Station2	Station3	Station4	Station5	Station6	Station7	Station8	Station9	Station10
Central	1	2	6	7	15	18	16	0	0	0
Northern	12	11	8	7	4	5	0	0	0	0
Circle	10	2	3	4	19	15	14	8	9	10
Victoria	13	11	9	6	4	19	17	0	0	0

Close the table when the data has been entered. Go to the Server Explorer window and delete the connection to the ***underground.mdf*** database.

We need to load the data for the underground lines when the program runs. Add a DataSet to hold the line data, and write a ***GetRailLines( )*** method. This uses very similar code to the method for loading the stations. You may save some time by copying code from ***GetStations( )***, then making the necessary changes.

```
DataSet dsStations = new DataSet();
DataSet dsRailLines = new DataSet();
public Form1()
{
    InitializeComponent();
    GetStations();
}
private void GetRailLines()
{
    try
    {
        SqlConnection con = new SqlConnection(@"Data Source=.\SQLEXPRESS;
             AttachDbFilename=" + databaseLocation + "Integrated Security=True;
             Connect Timeout=30; User Instance=True");
        con.Open();
        SqlCommand cmRailLines = new SqlCommand();
        cmRailLines.Connection = con;
        cmRailLines.CommandType = CommandType.Text;
        cmRailLines.CommandText = "SELECT * FROM lines";

        SqlDataAdapter daRailLines = new SqlDataAdapter(cmRailLines);
        daRailLines.Fill(dsRailLines);
        con.Close();
    }

    catch
    {
        MessageBox.Show("File error");
    }
}
```

When the program is running, we want the data for stations and underground lines to be easily available to the program so that a map of the Underground system can be drawn and journeys planned. It is best to hold the data in ***arrays*** in the fast electronic main memory of the computer, the RAM, where it can be accessed almost instantly.

The station data has already been stored in a set of arrays:

	StationID	StationName	Xpos	Ypos
[1]	1	Ealing Broadway	105	269
[2]	2	Notting Hill Gate	270	268
[3]	3	Paddington	272	221
[4]	4	Euston	409	218
.....	.....	.....	.....	.....
[19]	19	King's Cross	440	218

These are ***parallel*** arrays. This means that if we choose data from the same row in each array, the series of data items will all refer to the same station. For example, if we take ***row 4*** of each array, we know that the station with ***stationID = 4*** is ***Euston***, and its map cooordinates are ***Xpos = 409***, ***Ypos = 218***.

Notice that the arrays we created for the station data have only one column each. These are called ***one-dimensional arrays***.

When we transfer the Underground Lines data into arrays, a different structure will be needed. We can store the ***LineNames*** in a ***one-dimensional array***, but the station ID numbers along each line will need a ***two-dimensional array*** similar to a spreadsheet grid.

	LineName	Station 1	Station2	Station3	.....	Station 10
[1]	Central	1	2	6		0
[2]	Northern	12	11	8		0
[3]	Circle	10	2	3		10
[4]	Victoria	13	11	9		0

Once all the data is available in arrays, it is very easy for the computer to read the ***StationID*** numbers in sequence along each of the lines, then go to the station data to obtain the name and map coordinates for the station. For example: ***Station2*** on the ***Central*** line has ID number 2 and is therefore ***Notting Hill Gate***. The map coordinates for this station are ***Xpos = 270***, ***Ypos = 268***.

	LineName	Station 1	Station2	Station3	.....	Station 10
[1]	Central	1	2	6		0
[2]	Northern	12	11	8		0
[3]	Circle	10	2	3		10
[4]	Victoria	13	11	9		0

	StationID	StationName	Xpos	Ypos
[1]	1	Ealing Broadway	105	269
[2]	2	Notting Hill Gate	270	268
[3]	3	Paddington	272	221
[4]	4	Euston	409	218
.....	.....	.....	.....	.....
[19]	19	King's Cross	440	218

Set up variables near the start of the program to hold the ***names of the Underground Lines***, and the sets of ***stationID numbers along each line***. Add a call to the ***GetRailLines( )*** method:

```
DataSet dsRailLines = new DataSet();

int lineCount;
string[] lineName = new string[6];
int[,] stationNumber = new int[6, 12];

public Form1()
{
    InitializeComponent();
    GetStations();

    GetRailLines();
}
```

Go to the end of the ***GetRailLines( )*** method and add code to transfer the data into the arrays:

```
    catch

    {
        MessageBox.Show("File error");
    }

    lineCount = dsRailLines.Tables[0].Rows.Count;

    for (int i = 0; i < lineCount; i++)
    {
        DataRow drRailLine = dsRailLines.Tables[0].Rows[i];
        lineName[i] = Convert.ToString(drRailLine[0]);

        for (int j = 1; j < 11; j++)
        {
            stationNumber[i, j] = Convert.ToInt16(drRailLine[j]);
        }
    }
}
```

We now have a lot of variables in use in the program, and it is worth taking the time to construct a reference table, known as a ***Data Dictionary***, to remind ourselves of the purpose of each of the variables and the way that the array elements are accessed:

Variable	Data type	Purpose	Examples
stationCount	integer	The number of **station** records in the database	stationCount = 19
stationID	int[20]	One dimensional array storing the stationID numbers allocated to the stations.	stationID[1] = 1 stationID[19] = 19
stationName	string[20]	One dimensional array storing the names of the stations.	stationName[1]="Ealing Broadway" stationName[19]="Kings Cross"
stationX	int[20]	One dimensional array storing the X (across) pixel position of the station on the map	stationX[1] = 105 stationX[19] = 440
stationY	int[20]	One dimensional array storing the Y (down) pixel position of the station on the map	stationY[1] = 269 stationY[19] = 218
lineCount	int	The number of **underground line** records in the database	lineCount = 4
lineName	string[6]	One dimensional array storing the names of the underground lines.	lineName[1] = "Central"
stationNumber	int[6, 12]	Two dimensional array storing the stationID values for stations along each line. First index is the line number. Second index is the station sequence	stationNumber[2,3] = 8 (on underground line 2, the station in position 3 along the line has a stationID value of 8)

By use of the arrays it should be possible to find the sequence of stationIDs along any underground line, then use these stationID values to find the corresponding station names and map coordinates. We can therefore proceed to draw our route map:

Set up a method called ***DrawRailLines( )*** below the ***DrawMap( )*** method. Add a call to this method in ***Form1_Paint( )***.

We are going to draw the railway map as a series of straight line sections linking pairs of stations. Each line section will begin at the point (startX, startY) and end at the point (endX,endY).

We will set up pen colours to represent the offical colour codes given to the London Underground Lines:

Central Line:	***red***	***Northern Line:***	***black***
Circle Line:	***yellow***	***Victoria Line:***	***light blue***

The purpose of the FillRectangle command is to produce a white background for the map area.

```
private void DrawRailLines()
{
    int startX;
    int startY;
    int endX;
    int endY;

    Graphics g = this.CreateGraphics();
    Pen white = new Pen(Color.White, 1);
    Pen blackW = new Pen(Color.Black, 3);
    Pen goldW = new Pen(Color.Gold, 5);
    Pen blueW = new Pen(Color.DeepSkyBlue, 3);
    Pen redW = new Pen(Color.Red, 3);

    g.FillRectangle(white.Brush, new Rectangle(0, 0, 800, 600));
}

private void Form1_Paint(object sender, PaintEventArgs e)
{
    DrawRailLines();
    DrawMap();
}
```

We now add a loop to **DrawRailLines( )** which will repeat for each underground line, and within this a loop which will repeat for each pair of stations. We collect the stationIDs for the stations:

```
    Pen blueW = new Pen(Color.DeepSkyBlue, 3);
    Pen redW = new Pen(Color.Red, 3);
    g.FillRectangle(white.Brush, new Rectangle(0, 0, 800, 600));

    for (int i = 0; i < lineCount; i++)
    {
        for (int j = 1; j < 11; j++)
        {
            int firstStationID = stationNumber[i, j];
            int secondStationID = stationNumber[i, j + 1];
            if (secondStationID > 0)
            {
            }
        }
    }
```

We will use the stationIDs to find the map coordinates for each pair of stations, then connect them with a line of the correct colour. For example:

Central line

	Station 1	**Station 2**
	stationID = 1: **Ealing Broadway**	stationID = 2: **Notting Hill Gate**

map coordinates	Xpos = **105**, Ypos = **269**	Xpos = **270**, Ypos = **268**

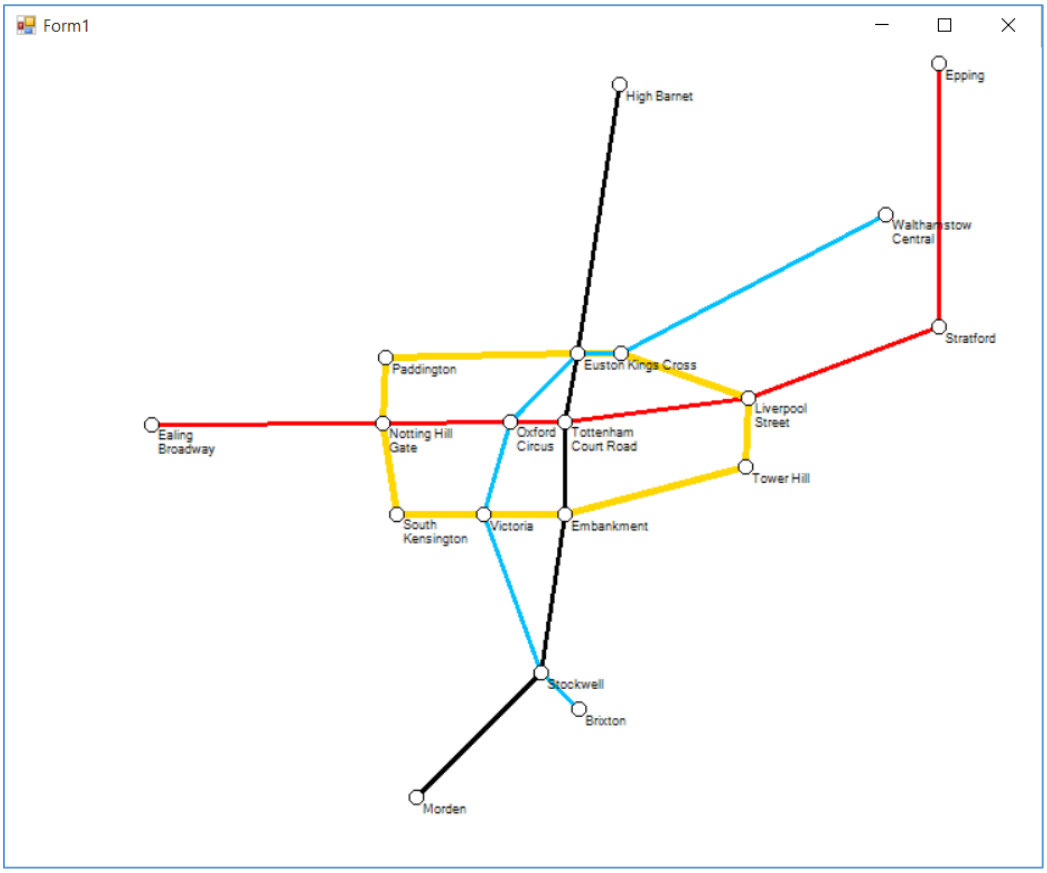
Add lines of code. These use the stationIDs to find the map coordinates for these stations:

```
for (int j = 1; j < 11; j++)
{
    int firstStationID = stationNumber[i, j];
    int secondStationID = stationNumber[i, j + 1];
    if (secondStationID > 0)
    {
        startX = 0;
        startY = 0;
        endX = 0;
        endY = 0;
        for (int s = 0; s < stationCount; s++)
        {
            if (firstStationID == stationID[s])
            {
                startX = stationX[s];
                startY = stationY[s];
            }
            if (secondStationID == stationID[s])
            {
                endX = stationX[s];
                endY = stationY[s];
            }
        }
    }
}
```

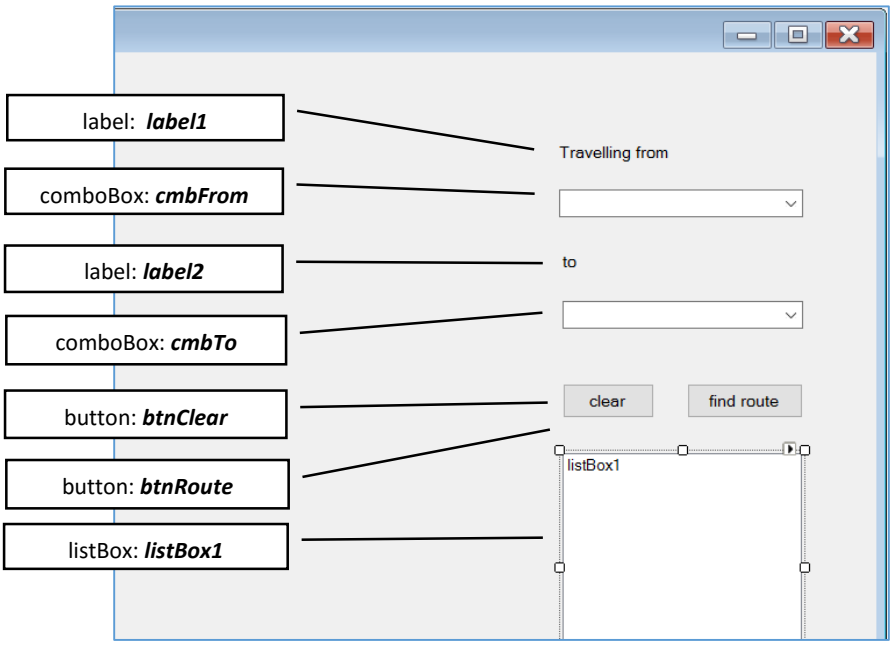
The final step is to draw a line between each pair of stations, using the correct colour for the particular London Underground Line:

```
    for (int s = 0; s < stationCount; s++)
    {

        if (firstStationID == stationID[s])
        {
            startX = stationX[s];
            startY = stationY[s];
        }
        if (secondStationID == stationID[s])
        {
            endX = stationX[s];
            endY = stationY[s];
        }
    }

    if (lineName[i] == "Circle")
        g.DrawLine(goldW, startX, startY, endX, endY);
    if (lineName[i] == "Northern")
        g.DrawLine(blackW, startX, startY, endX, endY);
    if (lineName[i] == "Victoria")
        g.DrawLine(blueW, startX, startY, endX, endY);
    if (lineName[i] == "Central")
        g.DrawLine(redW, startX, startY, endX, endY);
}
```

Run the program, and the railway map should be displayed with the stations connected as shown, using the correct line colours. If any stations are not connected correctly, go to the database to check that the stationID numbers have been allocated to stations correctly, and that the stationIDs appear in the correct sequence along each rail line.



Return to the Form1 design screen and add components to the top right hand corner of the form, beyond the edge of the map, as shown. You will need to extend the form sideways quite a long way to make space for this.



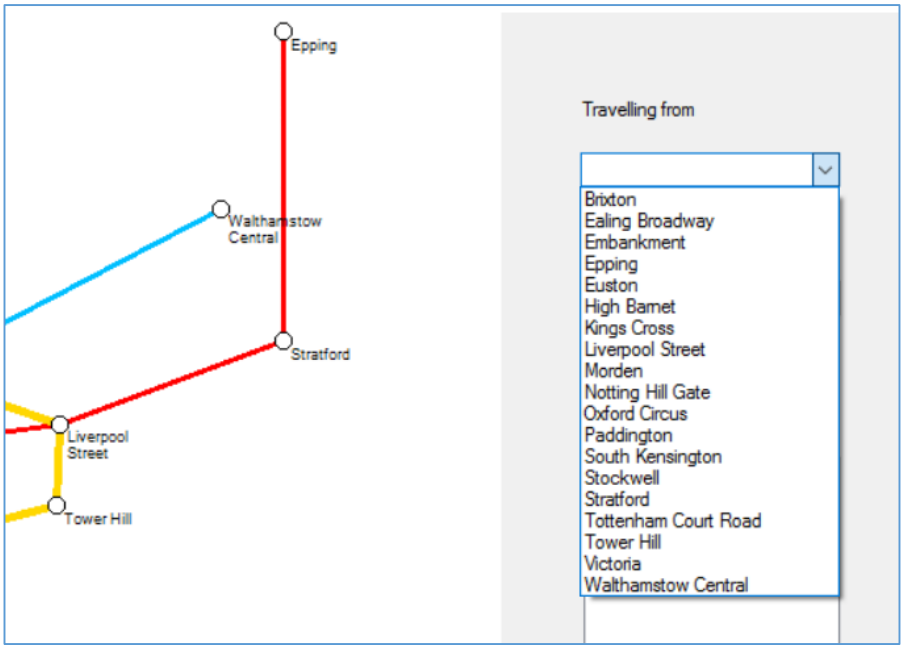
These components will provide a user interface for entering the start and destination stations, and for displaying the route found.

We will give the user the choice of selecting the start and destination stations from drop down alphabetical lists, or by clicking the required stations on the route map.

To produce the drop down lists, go to the ***GetStations( )*** method. Near the end of the method, add two lines of code to load the station lists into the comboBoxes:

```
        for (int i = 0; i < stationCount; i++)
        {
            DataRow drStation = dsStations.Tables[0].Rows[i];

            stationID[i] = Convert.ToInt16(drStation[0]);
            stationName[i] = Convert.ToString(drStation[1]);
            stationX[i] = Convert.ToInt16(drStation[2]);
            stationY[i] = Convert.ToInt16(drStation[3]);

            cmbFrom.Items.Add(stationName[i]);
            cmbTo.Items.Add(stationName[i]);
        }
    }
```

Run the program and check that the station names are listed in the comboBoxes and can be selected by mouse click. If necessary, move the components further to the right so they do not overlap the map.



A slight problem, easily corrected, is that the map redraws each time a ***comboBox*** is clicked, causing the screen to flicker. Go to the top of the program listing and add a ***Boolean*** (true/false) variable:

```
public partial class Form1 : Form
{
    string databaseLocation = "C:\\C#\\underground.mdf;";

    bool loading = true;
```

Add code to the ***Form1_Paint( )*** method to ensure that the map is only drawn once, at the time when the program first starts:

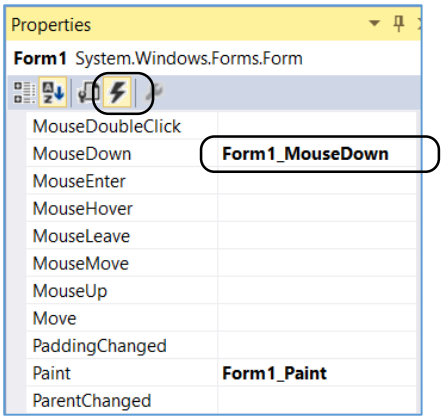
```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    if (loading == true)
    {
        DrawRailLines();
        DrawMap();
        loading = false;
    }
}
```

Double click the '***Clear***' button to create an on_click method, then add code to clear the entries in the comboBoxes and listBox.

```
private void btnClear_Click(object sender, EventArgs e)
{
    cmbFrom.Text = "";
    cmbTo.Text = "";
    listBox1.Items.Clear();
}
```

Run the program to check that the ***Clear*** button functions correctly.

We can now work on the code to select stations by clicking the route map. Begin by selecting Form1 and going to the Properties window. Change to '***Events***' and double click to create a '***MouseDown***' method:



Add code to the ***Form1_MouseDown( )*** method to find the X and Y position when the mouse is clicked on the form.

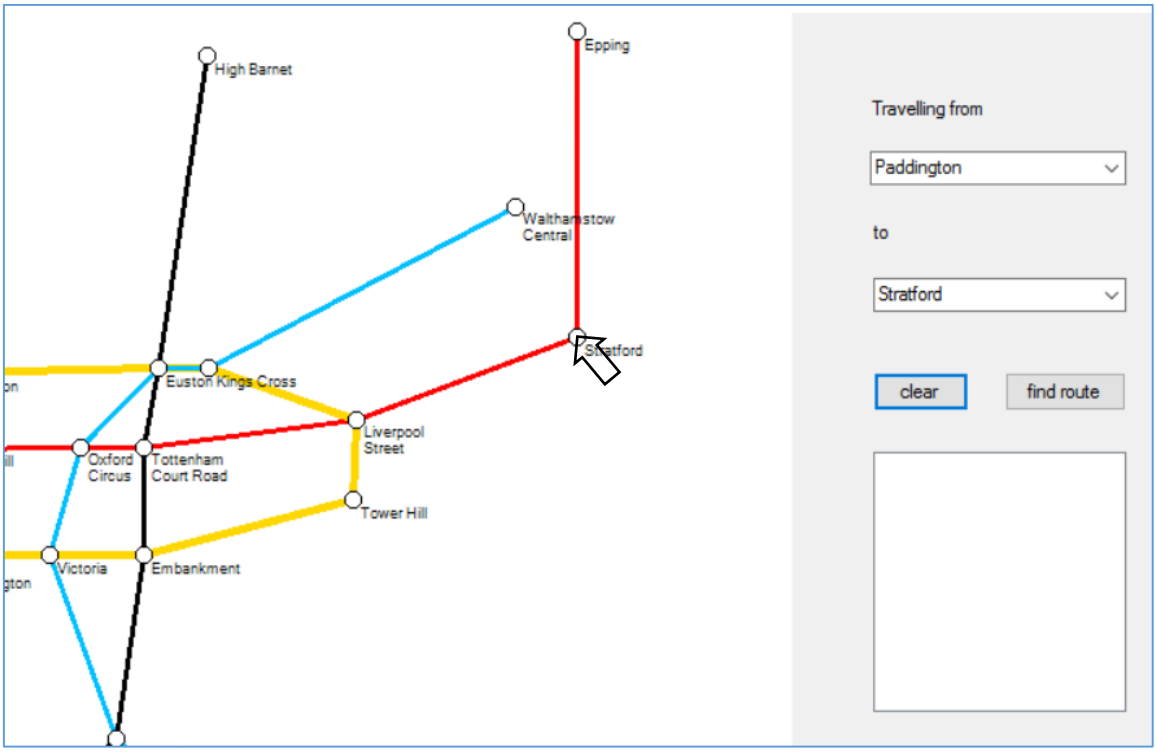
```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    int X = Cursor.Position.X;
    int Y = Cursor.Position.Y;
    Point p = new Point(X, Y);

    p = PointToClient(p);
}
```

We then use the X and Y cooordinates to look for a station close to this position on the map. The program checks for station cooordinates within 5 pixels of the mouse pointer when the mouse is clicked.

If the first comboBox, recording the starting location, is currently empty then the station name is entered in this box; otherwise it is entered into the second comboBox as the destination.

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    int X = Cursor.Position.X;
    int Y = Cursor.Position.Y;
    Point p = new Point(X, Y);
    p = PointToClient(p);
    for (int i = 0; i < stationCount; i++)
    {
        if (Math.Abs(stationX[i] - p.X) < 5 && Math.Abs(stationY[i] - p.Y) < 5)
        {
            if (cmbFrom.Text == "")
            {
                cmbFrom.Text = stationName[i];
            }
            else
            {
                cmbTo.Text = stationName[i];
            }
        }
    }
}
```

Run the program and check that stations can be selected correctly by clicking on the route map:

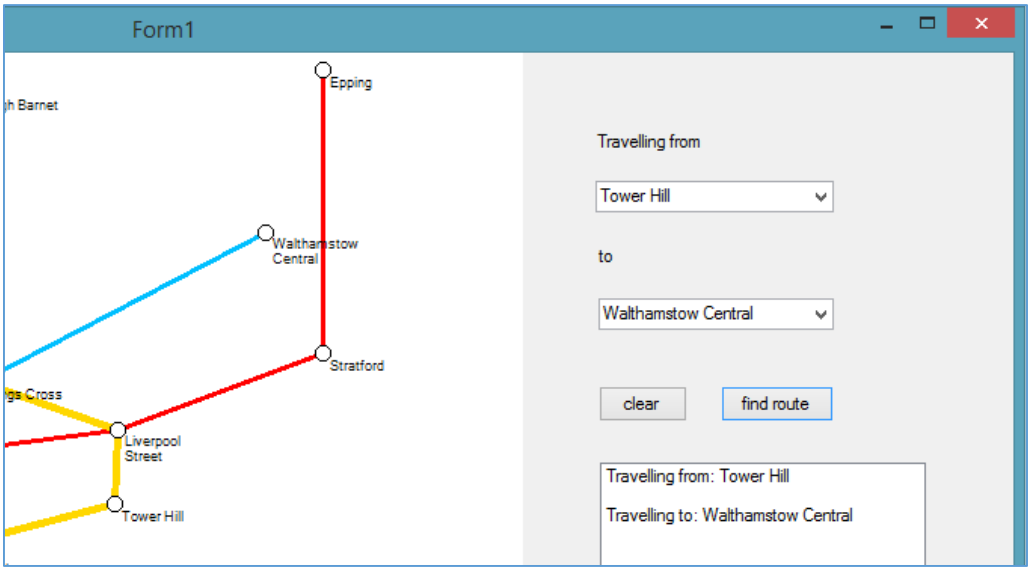


If the user interface is working correctly, we can now begin work on the route finding algorithm.

Close the program and go to the ***Form1*** design view. Double click the '***find route***' button to produce a ***btnRoute_Click( )*** method. Add lines of code to collect the names of the departure and destination stations from the comboBoxes and redisplay these in the listBox:

```
private void btnRoute_Click(object sender, EventArgs e)
{
    string startStation = cmbFrom.Text;
    string endStation = cmbTo.Text;

    listBox1.Items.Clear();
    listBox1.Items.Add("Travelling from: " + startStation);
    listBox1.Items.Add("");
    listBox1.Items.Add("Travelling to: " + endStation);
}
```

Run the program and check that the station names are transferred to the ListBox correctly. Increase the width of the List Box if necessary, so that the station names are fully visible.



Close the program and return to the C# code page.

We will set up some small methods which will be useful during the route finding process. The first will take the name of a station and ***convert it to the equivalent stationID*** number. Insert this method above the ***btnRoute_click( )*** method:

```
private int getStationID(string stationNameWanted)
{
    int stationIDfound = 0;
    for (int i = 0; i < stationCount; i++)
    {
        if (stationNameWanted == stationName[i])
        {
            stationIDfound = stationID[i];
        }
    }
    return stationIDfound;
}
```

Below this we will make another method to do exactly the opposite task, taking a stationID number and ***converting this to the equivalent station name***:

```
private string getStationName(int stationIDwanted)
{
    string stationNameFound = "";
    for (int i = 0; i < stationCount; i++)
    {
        if (stationIDwanted == stationID[i])
        {
            stationNameFound = stationName[i];
        }
    }
    return stationNameFound;
}
```

We need to add one more method to check whether a particular station is present on a particular underground line. This will return a result of '***true***' if the station is on the line, and '***false***' if it is not:

```
private bool stationOnLine(int lineNumber, int stationIDwanted)
{
    bool found = false;
    for (int j = 1; j < 11; j++)
    {
        if (stationNumber[lineNumber, j] == stationIDwanted)
        {
            found = true;
        }
    }

    return found;
}
```

Now that we have some useful tools available, we can continue with the route finding procedure.

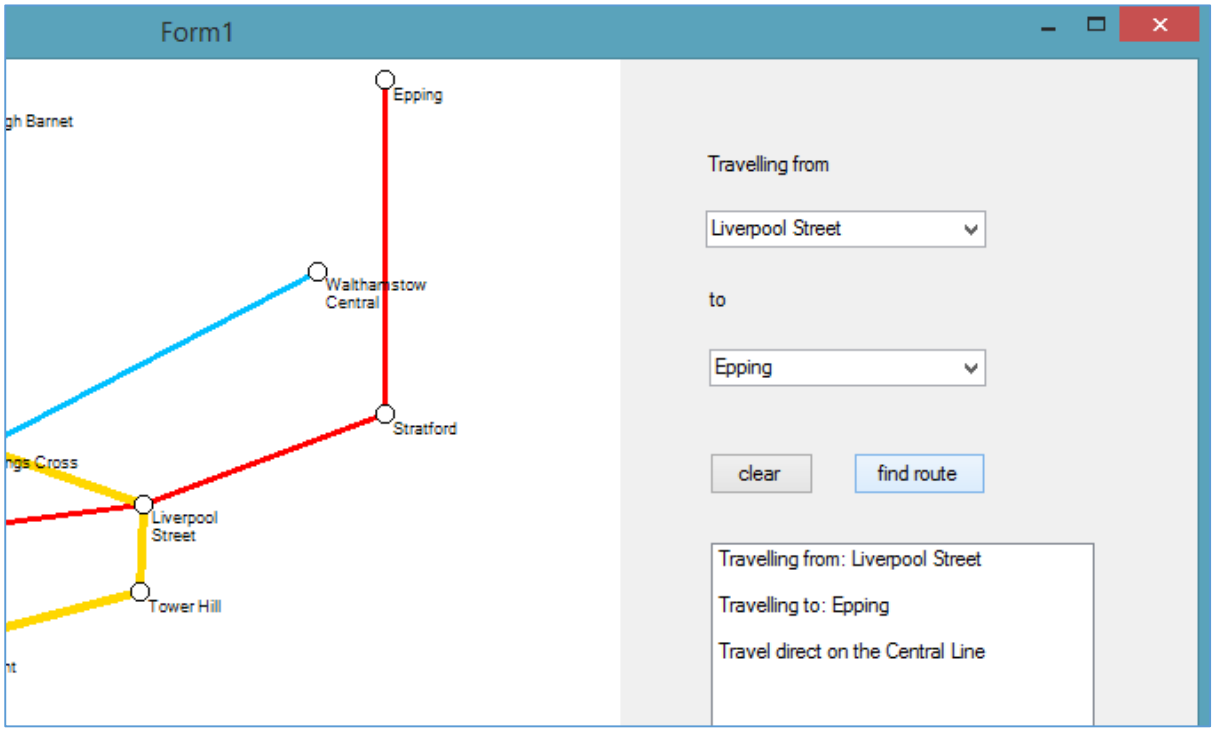
We know the names of the start and destination stations, so we can use the ***getStationID( )*** method to find the equivalent stationIDs. Add lines of code to the ***btnRoute_Click( )*** method.

```
listBox1.Items.Clear();
listBox1.Items.Add("Travelling from: " + startStation);
listBox1.Items.Add("");
listBox1.Items.Add("Travelling to: " + endStation);

int startStationID = getStationID(startStation);
int endStationID = getStationID(endStation);
```

If the stationIDs of both the start and destination stations are present on the same underground line, then it will be possible to make the journey without changing train. We will add code to the ***btnRoute_Click( )*** method to check for this possibility, making use of the ***stationOnLine( )*** method:

```
int startStationID = getStationID(startStation);
int endStationID = getStationID(endStation);

bool startFound, endFound;
string undergroundLine;
bool directRoute = false;

for (int lineNumber = 0; lineNumber < lineCount; lineNumber++)
{
    undergroundLine = lineName[lineNumber];

    startFound = stationOnLine(lineNumber, startStationID);
    endFound = stationOnLine(lineNumber, endStationID);

    if (startFound == true && endFound == true)
    {
        listBox1.Items.Add("");
        listBox1.Items.Add("Travel direct on the " + undergroundLine + " Line");
        directRoute = true;
    }
}
```

Run the program and test various routes where direct travel is possible:



We can now examine the more complicated situation where a change of train is necessary.

If no direct route is found then the following strategy will be adopted:

1.LOOP for each underground line

2. LOOP to find if the ***start station*** is on this line. IF it is…

3.LOOP for each station along the line – this is potentially a station to ***change*** trains

4.LOOP for each ***other*** underground line

5.IF the ***change station*** AND the ***destination station*** are on this line THEN
a route has been found:
Display the start line
Display the change station
Display the destination line

We will begin by adding ***loops 1 and 2***, to check each underground line to see if it ***contains the start station***:

```
        {
            listBox1.Items.Clear();
            listBox1.Items.Add("Travel direct on the "+undergroundLine+" Line");
            directRoute = true;
        }
    }
```

```
if (directRoute == false)
{

    for (int firstLine = 0; firstLine < lineCount; firstLine++)
    {
        string startUndergroundLine = lineName[firstLine];
        startFound = stationOnLine(firstLine, startStationID);
    }
}
```

If ***startFound*** is set to ***true***, then we have found an underground line serving the station where the traveller wishes to start their journey.

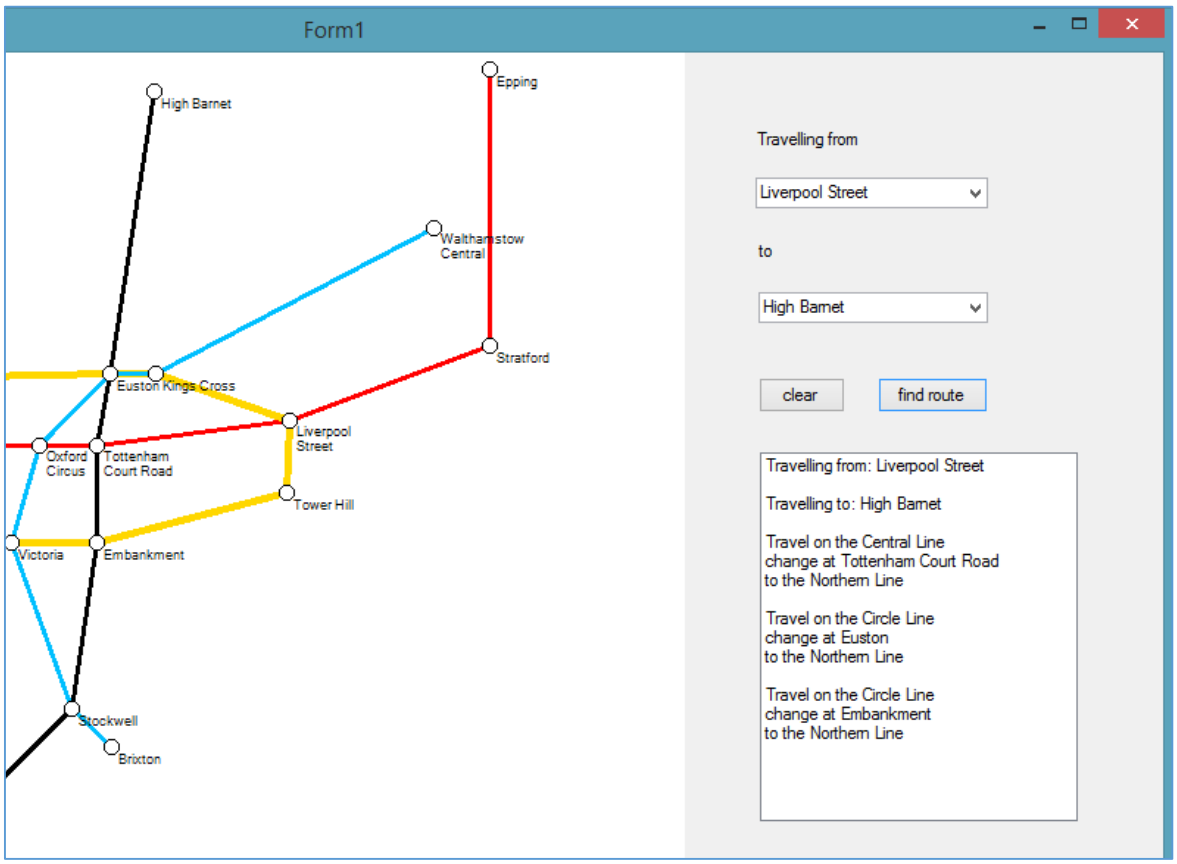
We will now consider each of the other stations along this line (***loop 3*** in the algorithm above), as it might be a ***possible point to change*** to a different underground line serving the destination.

```
for (int firstLine = 0; firstLine < lineCount; firstLine++)
{
    string startUndergroundLine = lineName[firstLine];
    startFound = stationOnLine(firstLine, startStationID);

    if (startFound == true)
    {
       for (int j = 1; j < 11; j++)
       {
           if (stationNumber[firstLine, j] > 0)
           {
               int changeStation = stationNumber[firstLine, j];
           }
       }
    }
}
```

We can now check each other underground line to see if the ***change station*** and ***destination*** are both on that line (***loops and conditionals 4 and 5*** in the algorithm above):

```
int changeStation = stationNumber[firstLine, j];

for (int secondLine = 0; secondLine < lineCount; secondLine++)
{
    if (secondLine != firstLine)
    {
         undergroundLine = lineName[secondLine];
         bool destination = stationOnLine(secondLine, endStationID);
         bool change = stationOnLine(secondLine, changeStation);
         if (destination==true && change==true)
         {
             string changeStationName=getStationName(changeStation);
             listBox1.Items.Add("");
             listBox1.Items.Add("Travel on the "+startUndergroundLine+" Line");
             listBox1.Items.Add("change at " + changeStationName);
             listBox1.Items.Add("to the " + undergroundLine + " Line");
         }
    }
}
```

Test the completed program, which should now give correct travel options between any starting station and destination. If more than one route is possible, each will be displayed.



Some programming challenges…

- If more than one route is possible, how could the program select the route through the least number of stations?
- How could the program be developed to include more stations and additional underground lines?
- For a more complex system, more than one change of train might be needed. How could the algorithm be developed to allow for two changes of train?