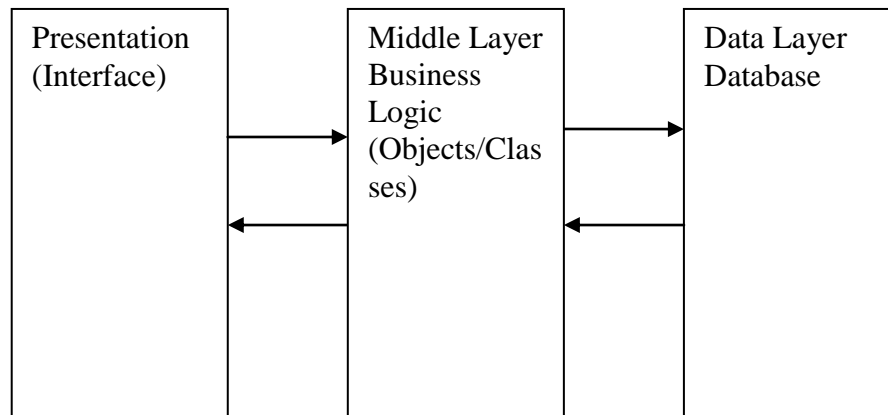


Copying the Data from Data Table to Array List

Cast your mind back to the second lecture to the following diagram.



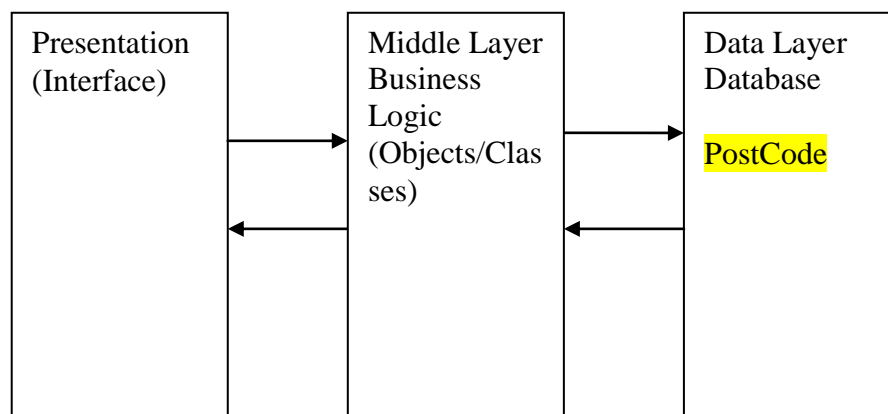
One thing we are trying to do is build our application around this three-layered architecture.

One design goal as we build our application is that the presentation layer must not have any knowledge of the data layer.

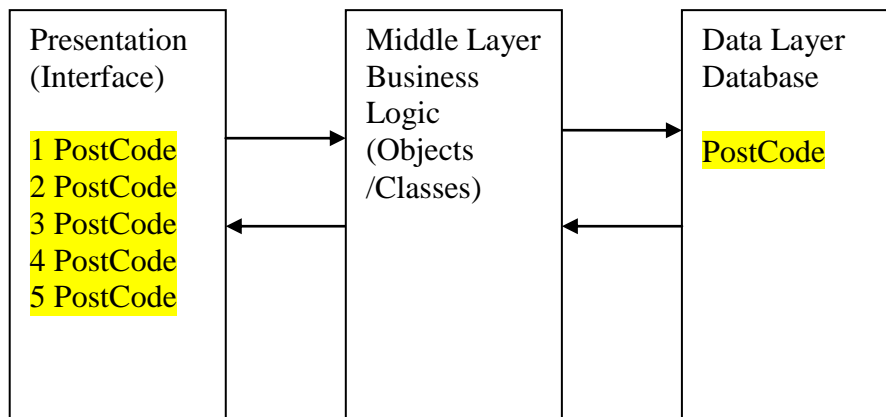
Why is this?

The reason why we do this is to think about what happens if we need to modify the database design.

Imagine in the data layer we have a field called PostCode.



Let us also imagine that we have 5 web pages that make use of this field directly linking to it.



The question then comes what happens if we re-name the `PostCode` field in the database to `PCode`?

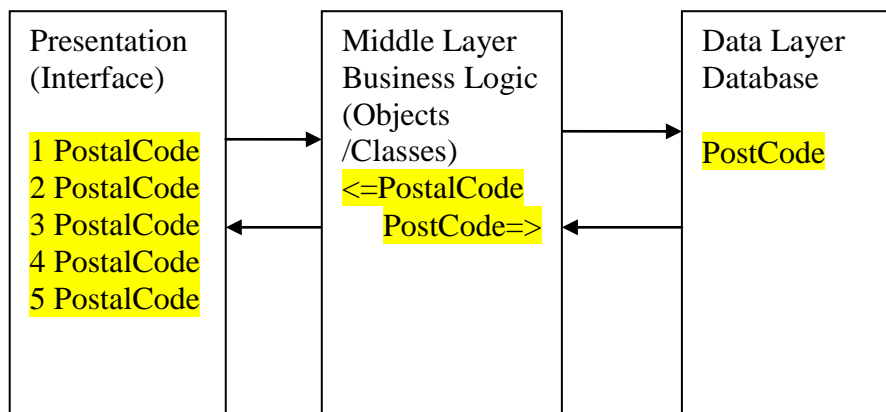
What we will have to do is make 5 additional changes in the presentation layer.

The way to get around this problem is to create an alias for the field that is defined in the middle layer.

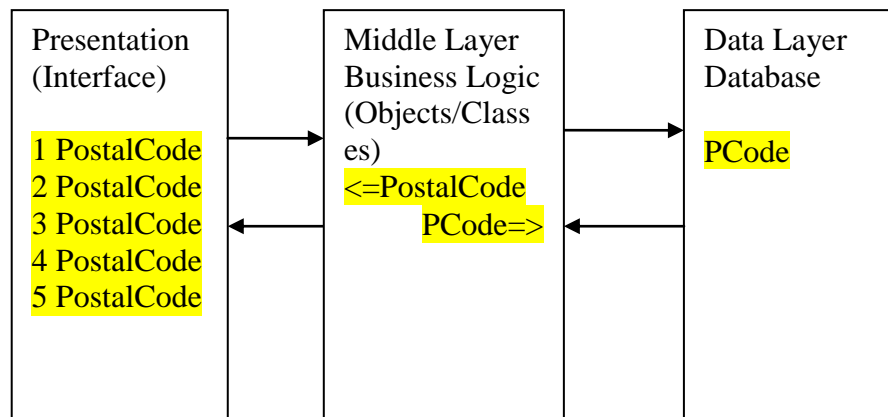
For this example, we shall call the alias `PostalCode`.

It is now the job of the middle layer to translate between presentation and data layers using this alias.

When the presentation layer wants to use the `PostCode` field it uses the alias `PostalCode` instead.



If we now change the name of the field in the data layer we change the name in the middle layer and the presentation layer carries on using the alias.



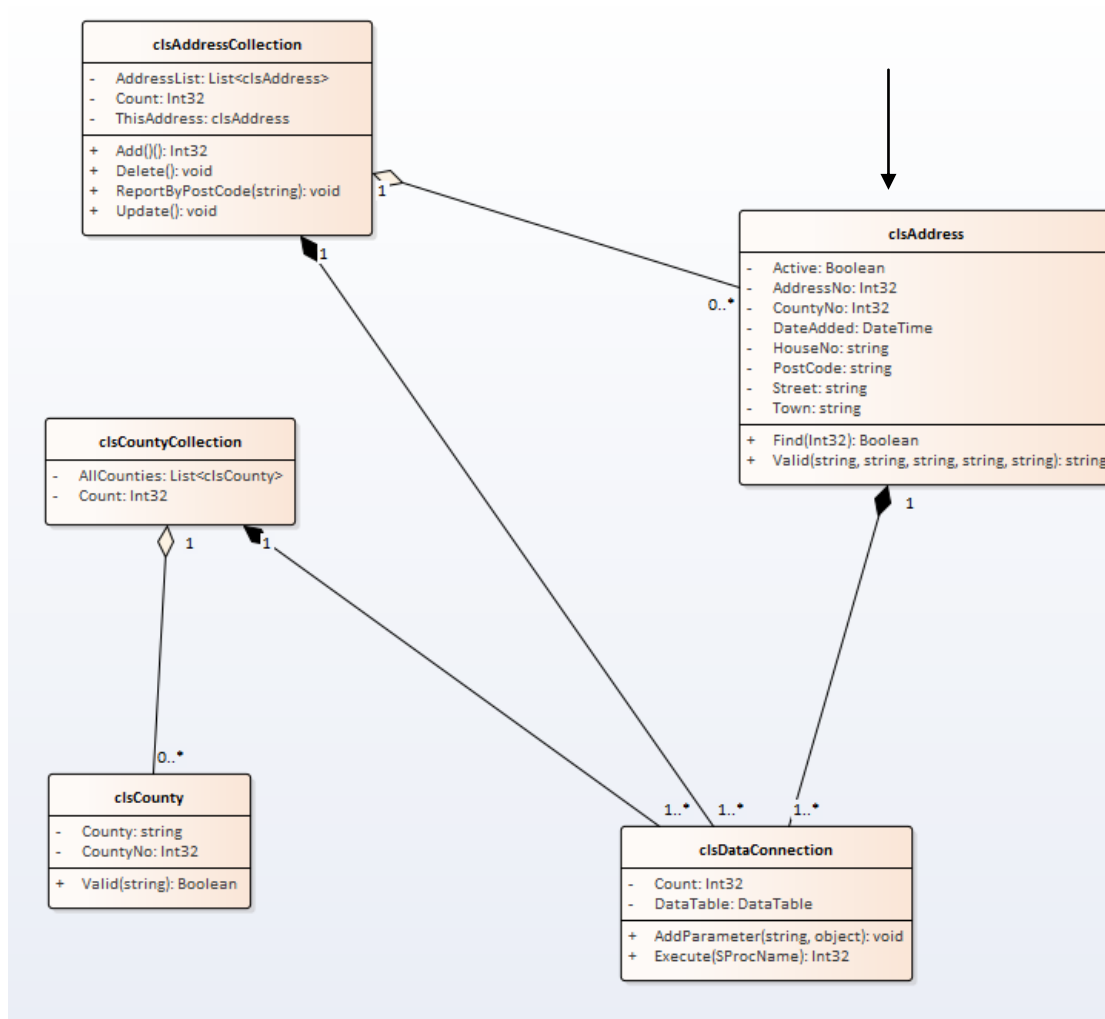
Encapsulation and Data Hiding

This example above shows how encapsulation may be used to hide the inner workings of an object. In this case, the presentation layer only uses the public methods and properties of the middle layer. It has no idea of how things really work in the middle layer and it is best if it doesn't.

To link the middle layer to the presentation layer we are going to have to do three things.

1. We will need to create a data storage class allowing us to create an array list of address book pages
2. We will then copy the data from the data table to this array list
3. We will lastly link the array list to the list box on the web form

We have already seen the data storage class in the class diagram. The class of interest is `clsAddress`.



When we look at a set of data such as that in a table it is quite complicated.

	AddressNo	HouseNo	Street	Town	PostCode	CountyCode	DateAdded	Active
	1	1	Some Street	Leicester	LE1 1BE	34	07/08/2013	True
	2	22	The Road	Nottingham	N19 6FF	48	07/08/2013	True
	3	33	High Street	Leicester	LE1 6FG	34	07/08/2013	True
▶*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

The best approach is to see the data as a grid which we navigate using the index and the field name.

[1][“Town”] gives us Nottingham
 [0][“DateAdded”] gives us 07/08/2013

Also think about how the data is made up so that we may think of the individual components.

The table in its entirety represents the full address book.



We already have a class allowing us to model the entire address book
clsAddressCollection the control class.

Individual rows in the data represent individual pages within the address book. To model this, we will need to create an address class.

We shall do that now.

Creating the Address Class

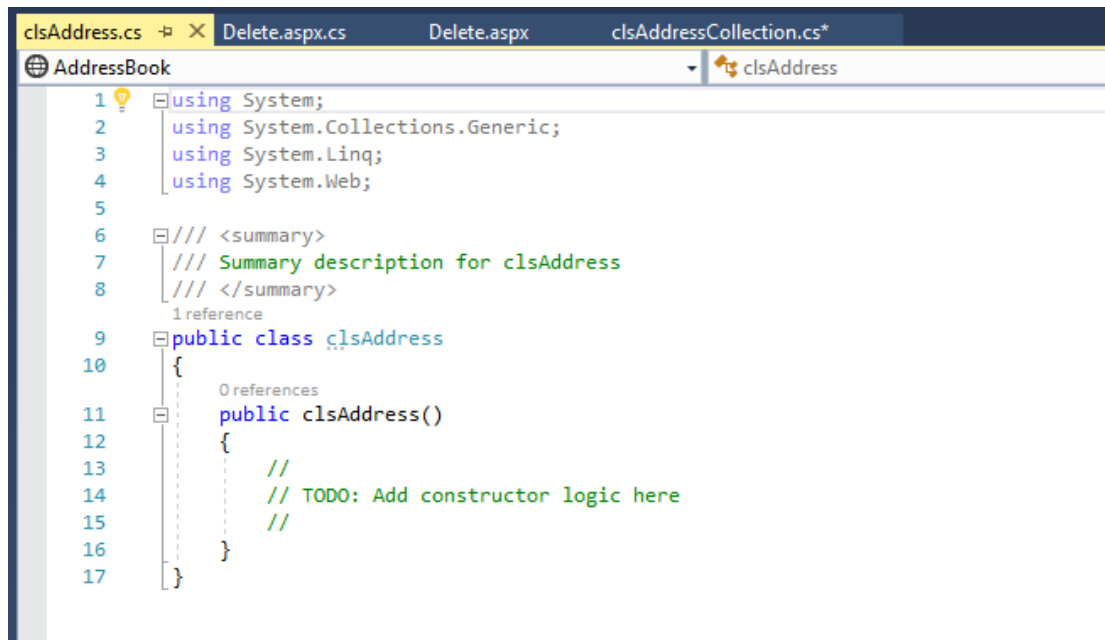
For the data storage class address page, it needs to map to the fields in the underlying data.

The table tblAddress has the following fields...

tblAddress	
AddressNo	Key
HouseNo	
Street	
Town	
PostCode	
CountyCode	
DateAdded	
Active	

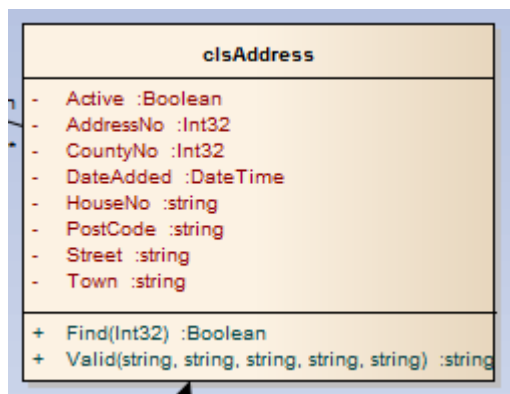
Each one of the fields needs to be mapped to a public property.

Create a new class definition called clsAddress ...



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Web;
5
6 /// <summary>
7 /// Summary description for clsAddress
8 /// </summary>
9 public class clsAddress
10 {
11     public clsAddress()
12     {
13         //
14         // TODO: Add constructor logic here
15         //
16     }
17 }
```

The address class will have two methods along with a set of properties reflecting the fields in the underlying table.



We also need to remember the rules of encapsulation! We want to create the inner workings of the class using private data-members and then only expose the essential features via public properties.

Functions for creating properties are a little more complicated than those we use for creating methods.

There are generally four parts to the property function.

```

//addressNo private member variable
private Int32 mAddressNo;
//AddressNo public property
public Int32 AddressNo
{
    get
    {
        //this line of code sends data out of the property
        return mAddressNo;
    }
    set
    {
        //this line of code allows data into the property
        mAddressNo = value;
    }
}

```

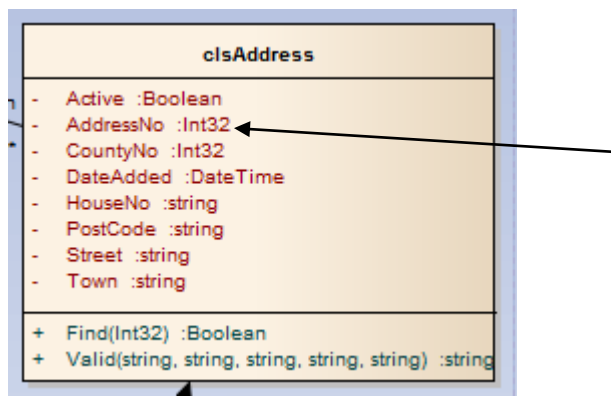
Private data member

Public property function

Getter

Setter

If we are to create the AddressNo property...



We would need to first create a variable to store the data.

```

//addressNo private member variable
private Int32 mAddressNo;

```

Notice the “m” prefix at the start of the variable. This reminds us (the programmer) this is a private data-member.

Following the rules of encapsulation and data hiding we only want the code in this class to have access to the data thus it is marked as “private”.

The next step is to create the property function...

```

//AddressNo public property
public Int32 AddressNo
{
    get
    {
    }
    set
    {
    }
}

```

```
}
```

There are two important parts to a property function.

The Getter allows data to be sent out of the public property to wherever it is needed. The Setter allows data to come into the public property to change the private data-member.

To do this we need to add a couple of extra lines of code.

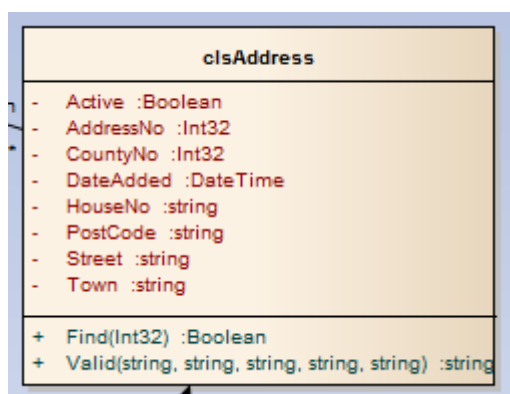
First the code that returns the private data.

```
//AddressNo public property
public Int32 AddressNo
{
    get
    {
        //this line of code sends data out of the property
        return mAddressNo;
    }
    set
    {
    }
}
```

Lastly, we need to add the code that allows us to change the private data...

```
//AddressNo public property
public Int32 AddressNo
{
    get
    {
        //this line of code sends data out of the property
        return mAddressNo;
    }
    set
    {
        //this line of code allows data into the property
        mAddressNo = value;
    }
}
```

The next step is to create property functions for all the attributes in the class.



Requires the following code...

```
//houseNo private member variable
private string mHouseNo;
//HouseNo public property
public string HouseNo
{
    get
    {
        return mHouseNo;
    }
    set
    {
        mHouseNo = value;
    }
}

//street private member variable
private string mStreet;
//Street public property
public string Street
{
    get
    {
        return mStreet;
    }
    set
    {
        mStreet = value;
    }
}

//town private member variable
private string mTown;
//Town public property
public string Town
{
    get
    {
        return mTown;
    }
    set
    {
        mTown = value;
    }
}

//postCode private member variable
private string mPostCode;
//PostCode public property
public string PostCode
{
    get
    {
        return mPostCode;
    }
    set
    {
        mPostCode = value;
    }
}
```

```

}

//countyCode private member variable
private Int32 mCountyCode;
//CountyCode public property
public Int32 CountyCode
{
    get
    {
        return mCountyCode;
    }
    set
    {
        mCountyCode = value;
    }
}

//dateAdded private member variable
private DateTime mDateAdded;
//DateAdded public property
public DateTime DateAdded
{
    get
    {
        return mDateAdded;
    }
    set
    {
        mDateAdded = value;
    }
}

//active private member variable
private Boolean mActive;
//Active public property
public Boolean Active
{
    get
    {
        return mActive;
    }
    set
    {
        mActive = value;
    }
}

//addressNo private member variable
private Int32 mAddressNo;
//AddressNo public property
public Int32 AddressNo
{
    get
    {
        //this line of code sends data out of the property
        return mAddressNo;
    }
    set
    {
        //this line of code allows data into the property
        mAddressNo = value;
    }
}

```

```
}
```

Getters and Setters – How they work

To understand how this works we need to return to our old friend the assignment operator.

We have seen on many occasions how we may use the assignment statement to do things such as copy data from a property in an interface control to the RAM...

```
//copy the value entered on the interface into the RAM
AddressNo = txtAddressNo.Text;
```

And back again...

```
//display an error
lblError.Text = SomeError;
```

To get data in and out of the property we need a route where the data comes in and a route where the data is sent out.

In this class, we have a private member variable called street...

```
//street private member variable
private string mStreet;
```

The fact that this is set to private means that we cannot get at it outside of this class e.g. in the presentation layer!

To access the private data, we send it values via the setter and get its value via the getter like so...

```
//Street public property
public string Street
{
    get
    {
        return mStreet;
    }
    set
    {
        mStreet = value;
    }
}
```

To see this working in more detail look at the following section of code...

```
//create an object based on the address class
clsAddress AnAddress = new clsAddress();
//set the Street property
AnAddress.Street = "Some Street";
```

This code creates an object based on the class definition clsAddress.

When the assignment operator copies the data, the data passes into the class via the properties setter...

```
//Street public property
public string Street
{
    get
    {
        return mStreet;
    }
    set
    {
        mStreet = value;
    }
}
```

The value of the parameter “value” is set to “Some Street” and assigned to the private data-member mStreet.

If we extend the code segment like so...

```
//create an object based on the address class
clsAddress AnAddress = new clsAddress();
//set the Street property
AnAddress.Street = "Some Street";
//declare a variable to store the street
string TheStreet;
//copy the data from the middle layer to the RAM
TheStreet = AnAddress.Street;
```

We now declare a variable called TheStreet.

The assignment operator copies the data in the property Street. To get at the data it accesses the properties Getter

```
//Street public property
public string Street
{
    get
    {
        return mStreet;
    }
    set
    {
        mStreet = value;
    }
}
```

Since in the previous section of code street was assigned the value “Some Street” the getter will return this value to the assignment operator.

```
//copy the data from the middle layer to the RAM
```

```
TheStreet = AnAddress.Street;
```

TheStreet will therefore be assigned the value of “Some Street”.

Now that we have created a class allowing us to store data for a single address book page we will create an array list to show how this works.

Creating the Array List

Open the code for clsAddressCollection and locate the function AddressList...

```
public void AddressList()
{
    //declare variables for each field in the table
    Int32 AddressNo;
    string HouseNo;
    string Town;
    string Street;
    string PostCode;
    Int32 CountyCode;
    DateTime DateAdded;
    Boolean Active;
    //var to store the count of records
    Int32 RecordCount;
    //var to store the index for the loop
    Int32 Index = 0;
    //create a connection to the database
    clsDataConnection dBConnection = new clsDataConnection();
    //send a post code filter to the query
    dBConnection.AddParameter("@PostCode", "");
    //execute the query
    dBConnection.Execute("sproc_tblAddress_FilterByPostCode");
    //get the count of records
    RecordCount = dBConnection.Count;
    //keep looping till all records are processed
    while (Index < RecordCount)
    {
        //copy the data from the table to the RAM
        AddressNo = Convert.ToInt32(dBConnection.DataTable.Rows[Index]["AddressNo"]);
        HouseNo = Convert.ToString(dBConnection.DataTable.Rows[Index]["HouseNo"]);
        Town = Convert.ToString(dBConnection.DataTable.Rows[Index]["Town"]);
        Street = Convert.ToString(dBConnection.DataTable.Rows[Index]["Street"]);
    }
}
```

So far, the loop is working through each record in the data table but data isn't going anywhere.

We want to get the data to the presentation layer so we need to create the “alias” for it using an array list.


The first step is to create an array list of type clsAddress.

```
References
public void AddressList()
{
    //create an array list of type clsAddress
    List<clsAddress> mAddressList = new List<clsAddress>();
    //declare variables for each field in the table
    Int32 AddressNo;
```

This gives us somewhere to copy the data from the data table to.

The next step is to create a “NewAddress” object at the start of the loop.

```
RecordCount = dBConnection.Count;
//keep looping till all records are processed
while (Index < RecordCount)
{
    //create a blank address page
    clsAddress NewAddress = new clsAddress();
    //copy the data from the table to the RAM
    NewAddress.AddressNo = Convert.ToInt32(dBConnection.Da
    NewAddress.HouseNo = Convert.ToString(dBConnection.Dat
    NewAddress.Town = Convert.ToString(dBConnection.DataTa
    NewAddress.Street = Convert.ToString(dBConnection.Dat
```




Now, rather than copying the data to the variables we copy the data to the properties in the new object NewAddress...

```
RecordCount = dBConnection.Count;
//keep looping till all records are processed
while (Index < RecordCount)
{
    //create a blank address page
    clsAddress NewAddress = new clsAddress();
    //copy the data from the table to the RAM
    NewAddress.AddressNo = Convert.ToInt32(dBConnection.DataTable.Rows[Index]["AddressNo"]);
    NewAddress.HouseNo = Convert.ToString(dBConnection.DataTable.Rows[Index]["HouseNo"]);
    NewAddress.Town = Convert.ToString(dBConnection.DataTable.Rows[Index]["Town"]);
    NewAddress.Street = Convert.ToString(dBConnection.DataTable.Rows[Index]["Street"]);
    NewAddress.PostCode = Convert.ToString(dBConnection.DataTable.Rows[Index]["PostCode"]);
    NewAddress.CountyCode = Convert.ToInt32(dBConnection.DataTable.Rows[Index]["CountyCode"]);
    NewAddress.DateAdded = Convert.ToDateTime(dBConnection.DataTable.Rows[Index]["DateAdded"]);
    NewAddress.Active = Convert.ToBoolean(dBConnection.DataTable.Rows[Index]["Active"]);
    //add the blank page to the array list
    mAddressList.Add(NewAddress);
    //increase the index
    Index++;
}
```

(We can also get rid of the variables now we have our data storage object in place.)

Once we have the new address set up with the data we may add it to the array list.

```
//keep looping till all records are processed
while (Index < RecordCount)
{
    //create a blank address page
    clsAddress BlankPage = new clsAddress();
    //copy the data from the table to the RAM
    BlankPage.AddressNo = Convert.ToInt32(dBConnection.DataTable.Rows[Index]["AddressNo"]);
    BlankPage.HouseNo = Convert.ToString(dBConnection.DataTable.Rows[Index]["HouseNo"]);
    BlankPage.Town = Convert.ToString(dBConnection.DataTable.Rows[Index]["Town"]);
    BlankPage.Street = Convert.ToString(dBConnection.DataTable.Rows[Index]["Street"]);
    BlankPage.PostCode = Convert.ToString(dBConnection.DataTable.Rows[Index]["PostCode"]);
    BlankPage.CountyCode = Convert.ToInt32(dBConnection.DataTable.Rows[Index]["CountyCode"]);
    BlankPage.DateAdded = Convert.ToDateTime(dBConnection.DataTable.Rows[Index]["DateAdded"]);
    BlankPage.Active = Convert.ToBoolean(dBConnection.DataTable.Rows[Index]["Active"]);
    //add the blank page to the array list
    mAddressList.Add(BlankPage);
    //increase the index
    Index++;
}
```



Use the debugger to check that the function works correctly.

We have almost completed this section of the work. The last step is to link the array list in the middle layer to the list box in the presentation layer.