6 Using Databases on Windows Phone

6.1 An Overview of Database Storage

A database is a collection of, well, data, which is organised and managed by a computer program which is often, and rather confusingly, called a database.

Another computer program can ask a database questions and the database will come back with the results. You don't think of a database as part of your program as such, it is the component in your solution that deals with data storage.

We could discover how a database works by taking a quick look at a very simple example. Consider how we would create a database to hold information for an internet shop. We have a large number of customers that will place orders for particular products.

If we hire a database designer they will charge us a huge amount of money and then they will come up with some designs for database tables that will hold the information in our system. Each of the tables will have a set of columns that hold one particular property of an item, and a row across the table will describe a single item. This is the design for the customer table:

Customer ID	Name	Address	Bank Details
123456	Rob	18 Pussycat Mews	Nut East Bank
654322	Jim	10 Motor Drive	Big Fall Bank
111111	Ethel	4 Funny Address	Strange bank

This is a **Customers** table that our sales system will use. It contains the same information that was stored in the Customer Manager, with the addition of a string which gives bank details for that customer. Each row of the table will hold information about one customer. The table could be a lot larger than this; it might also hold the email address of the customer, their birthday and so on.

Product ID	Product Name	Supplier	Price
1001	Windows Phone 7	Microsoft	200
1002	Cheese grater	Cheese Industries	2
1003	Boat hook	John's Dockyard	20

This is the **Products** table. Each row of this table describes a single product that the store has in stock. This idea might be extended so that rather than the name of the supplier the system uses a Supplier ID which identifies a row in the Suppliers table.

Order ID	Customer ID	Product ID	Quantity	Order Date	Status
1	123456	1001	1	21/10/2010	Shipped
2	111111	1002	5	10/10/2010	Shipped
3	654322	1003	4	1/09/2010	On order

This is the **Orders** table. Each row of the table describes a particular order that has been placed. It gives the Product ID of the product that was bought, and the Customer ID of the customer that bought it.

By combining the tables you can work out that Rob has bought a Windows phone, Ethel has bought five Cheese graters and the four Boat hooks for Jim are on order.

This is a bit like detective work or puzzle solving. If you look in the Customer table you can find that the customer ID for Rob is 123456. You can then look through the order table and find that customer 123456 ordered something with an ID of 1001. You can then look through the product table and find that product 1001 is a Windows Phone (which is actually not that surprising).

Databases and Queries

You drive a database by asking it questions, or queries. We could build a query that would find all the orders that Rob has placed. The database system would search through the Orders table and return all the rows that have the Customer ID of 123456. We could then use this table to find out just what products had been purchased by searching through the Products table for each of the Product IDs in the orders that have been found. This would return another bunch of results that identify all the things I have bought. If I want to ask the database to give me all the orders from Rob I could create a query like this:

```
SELECT * FROM Orders WHERE CustomerID = "123456"
```

This would return a “mini-table” that just contained rows with the Customer ID of “123456”, i.e. all the orders placed by Rob. The commands I’m using above are in a language called SQL or Structured Query Language. This was specifically invented for asking questions of databases.

Companies like Amazon do this all the time. It is how they manage their enormous stocks and huge number of customers. It is also how they create their “Amazon has recommendations for you” part of the site, but we will let that slide for now.

Connecting to a Database

Unfortunately object oriented programs do not work in terms of tables, rows and queries. They work in terms of objects that contain data properties and methods. When a program connects to a database we need a way of managing this transition from tables to objects.

Databases and Classes

We have already seen that we can represent data in classes. The `Customer` class in the Customer Manager application we have been working on holds information about each customer:

```
public class Customer
{
    public int CustomerID { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string BankDetails { get; set; }

    public Customer(int inID, string inName, string inBank,
                    string inAddress,)
    {
        Name = inName;
        Address = inAddress;
        BankDetails = inBank;
        ID = inID;
    }
}
```

To hold a large number of customers we created a class that holds a list of them.

```
List<Customer> Customers = new List<Customer>();
```

Whenever I add a new customer I add it to the `Customers` list and to find customers I use the `foreach` loop to work through the customers and locate the one I want. As an example, to find all the orders made by a customer I could do something like this:

```
public List<Order> FindCustomerOrders(int CustomerID)
{
    List<Order> result = new List<Order>();

    foreach ( Order order in Orders )
    {
        if (order.CustomerID == CustomerID)
        {
            result.Add(order);
        }
    }
    return result;
}
```

The method searches through all the orders and builds a new list containing only ones which have the required customer ID. It is a bit more work than the SQL query, but it has the same effect.

Using LINQ to connect Databases to Objects

People like using databases because it is easier to create a query than write the code to perform the search. However, they also like using objects because they are a great way to structure programs. To use databases and object oriented programs together the programmer would usually have to write lots of “glue” code that transferred data from SQL tables into objects and then back again when they are stored. This is a big problem in large developments where they may have many data tables and classes based on information in the tables.

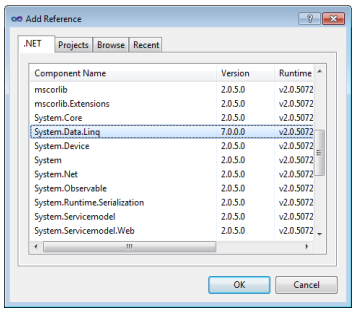
Language Integrated Query, or LINQ, was the result of an effort to remove the need for all this glue. It is called “Language Integrated” because they actually had to change the design of the C# language to be able to make the feature work.

Windows Phone supports the creation of databases for our programs to use and all the database interaction is performed using LINQ. So now we are going to see how we can take our class design and use it to create a database. We are going to use SQL to store the customers in our Customer Manager. Then we are going to add support for product orders that they place. This will let our user hold a large number of customers, orders and products in a database on the phone.

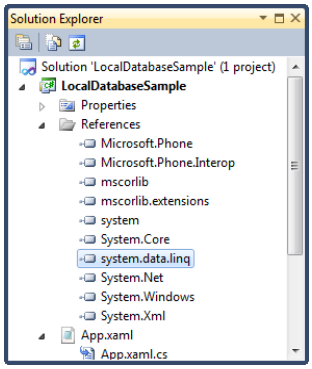
The next thing we are going to do is start to design the database tables, but before we can do that we need to make LINQ work with this project.

The LINQ Libraries

Before a program can use the LINQ code we have to add a reference to the `System.Data.Linq` libraries to the project. Because not all programs need database support, this reference is normally left out when a new project is created. We can open the Add Reference dialog by selecting Add Reference from the Project menu in Visual Studio. This gives us a list of libraries from which we can select the one we want.



When OK is clicked the reference is added to the list of resources that the program requires.



The next thing we should do is add some “using” directives to make it easier to access the classes in this library.

```
using System.Linq;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.ComponentModel;
using System.Collections.ObjectModel;
```

We can now use all the classes from these libraries without having to use the fully qualified form of each name.

Creating a LINQ Table

Next we want to create some table designs that our database is going to manage. There will be three tables in our database.

- Customers
- Orders
- Products

We could start by considering how we are going to store information about a customer. From the original table we created a `Customer` class that describes the data a customer must contain. Now we want to use this class to create a design for a database table that will hold all our customers. We can do this by taking our original `Customer` class and modifying it so that LINQ can use the class design to create a table in the database from it. This is our `Customer` class design.

```
public class Customer
{
    public string Name {get; set;}
    public string Address { get; set; }
    string BankDetails { get; set; }
    public int CustomerID { get; set; }
}
```

This class holds three strings and an integer, which are all members of the class. They are implemented as properties, with **get** and **set** behaviours. The start of our Customer class for LINQ will look rather similar:

```
[Table]
public class Customer : INotifyPropertyChanged,
                        INotifyPropertyChanging
{
    // Customer table design goes here
}
```

The [Table] item is an *attribute*. Attributes are used to flag classes with information which can be picked up by programs that look at the *metadata* in the compiled code. Metadata, as I am sure you are all aware, is "data about data". In this case the data being added is the attribute, and the data it is about is the Customer class.

When a C# class is compiled the compiler adds lots of metadata to the output that is produced, including the precise version of the code, details of all the methods it contains and so on. Other programs can read this metadata, along with the class information. This how ildasm produced the output we saw in chapter 3 when we inspected the contents of the files produced by Visual Studio.

The code above adds the [Table] attribute to the Customer class, which LINQ interprets as meaning "This class can be used as the basis of a data table". There is actually nothing much inside the [Table] attribute itself, it just serves as a tag.

The code above also states that Customer class implements the INotifyPropertyChanged and INotifyPropertyChanging interfaces. A class implements an interface when it contains all the methods that are specified in the interface. The two interfaces contain methods that will be used to tell LINQ when the content of the data in the class are being changed.

We have seen this situation before, where data bound to a Silverlight display element needed to communicate changes so that the display updated automatically when the data was changed. This situation is very similar, except that a change to the data is going to trigger updates in the database.

Each interface contains a single event delegate. This is the delegate for INotifyPropertyChanged.

```
public event PropertyChangedEventHandler PropertyChanged;
```

The LINQ infrastructure will bind to the PropertyChanged event so that it can be told when a property value has changed. There is also a delegate to be used to tell LINQ when a value is changing. This event is fired before the change is executed:

```
public event PropertyChangingEventHandler PropertyChanging;
```

Our job is to make sure that the Customer class fires these events when data changes. In other words, if a programmer does something like this:

```
activeCustomer.Name = "Trevor";
```

- the code that updates the Name property should also tell LINQ that the data has changed. This means that the Name property could look like this:

```
private string nameValue;

public string Name
{
    get
    {
        return nameValue;
    }
    set
    {
        if (PropertyChanging != null)
        {
            PropertyChanging(this,
                  new PropertyChangingEventArgs("Name"));
        }
        nameValue = value;
        if (PropertyChanged != null)
        {
            PropertyChanged(this,
                  new PropertyChangedEventArgs("Name"));
        }
    }
}
```

The `Get` behaviour for the property just returns the value of the data member that holds the name.

The `Set` behaviour tests to see if anything is attached to the event handlers for the events and calls one before the property is changed, and the other after. The delegate calls are supplied with two parameters. The first is **`this`** , which is a reference to the currently active `Customer` instance (the one whose name is changing). The second is an event argument that contains the name of the property that is being changed. LINQ can use these to work out what has changed and perform an appropriate database update. As we saw when using Silverlight, if the name of the property is specified incorrectly (we say “name” or “nayme”) the update will not work correctly.

The change generates “before” and “after” events so that LINQ can manage the data changes more efficiently. If this seems like a lot of extra work, remember that the result of our efforts will be that when we update objects from our database we don’t have to worry about storing the effects of our changes. This will all happen automatically.

If our objects contain a lot of data fields it makes sense to simplify the changed event management by writing some methods that do the notification management for us:

```
private void NotifyPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this,
                    new PropertyChangedEventArgs(propertyName));
    }
}

private void NotifyPropertyChanging(string propertyName)
{
    if (PropertyChanging != null)
    {
        PropertyChanging(this,
                    new PropertyChangingEventArgs(propertyName));
    }
}
```

We can then simplify the name property to this behaviour:

```
private string nameValue;

public string Name
{
    get
    {
        return nameValue;
    }
    set
    {
        NotifyPropertyChanging("Name");
        nameValue = value;
        NotifyPropertyChanged("Name");
    }
}
```

If this `NotifyPropertyChanged` stuff seems rather tiresome, remember why we are doing it. We are doing it so that we can write code such as:

```
activeCustomer.Name = "Fred Blogs";
```

- and have the database entry for the currently active customer automatically update. This is really useful. If we make use of TwoWay data binding the result will be that once we have connected our data objects to the display elements they will update themselves automatically, which is very useful indeed.

There is one final thing we need to add to the `Name` property to allow it to be used by LINQ. We have to mark the property with the **`[Column]`** attribute, so that LINQ knows that it is to be used in the database:

```
[Column]
public string Name
{
    // Name property behaviour goes here.
}
```

The address and bank details items can be added to the **`Customer`** class in exactly the same way. Each of them will map onto another column in the database.

Creating a Primary Key

The CustomerID item is slightly different however. In our original Customer Manager it was an integer value that identified each customer. If the customer gets married and changes their name the customer ID is there to make sure we can still find them in the application. In database terms we are using the Customer ID as the *primary key* for this table. The primary key is used to uniquely identify particular customers in this table. We might have two customers with the name “John Smith”, but we will **never** have two customers with the same ID. The ID works in exactly the same way as your Social Security number or bank account number.

When we create the ID column we have to tell LINQ that this is the primary key for the table. We can also ask LINQ to make sure that all values in this column are unique, and we can even ask that the values for this element be auto-generated. We do this by adding information to the Column attribute for the CustomerID property:

```
[Column(IsPrimaryKey = true, IsDbGenerated = true)]

public int CustomerID { get; set; }
```

When we add a new customer entry to the database we don’t have to worry about the ID, as it will be generated for us. We can of course view the ID value, so that we can tell the next “John Smith” what their unique id is.

Note that we are **never** allowed to change the ID for a customer. This is a unique key which will be automatically created by the database for each customer in turn. In database terms this property is called the *primary key* for the customer record. The

primary key of a customer will be used in our database to implement a relationship between the Customers table and the Orders table. Each row of the Orders database will contain a customer ID value that identifies the customer who placed that order.

When LINQ reads the metadata for this property it will use these settings to decide what kind of database column to create.

Creating a LINQ Data Context

Now that we have the design for a row in the table we can create the table itself. In our previous customer manager application we used a List to hold the customers. In LINQ we create a `DataContext` which will manage the connection to the database we are about to create. This will contain all the tables in the application, which at the moment just means the customers:

```
public class SalesDB : DataContext
{
    public Table<Customer> Customers;

    public Customers(string connection) : base(connection)
    {
    }
}
```

The `SalesDB` class extends the `DataContext` class, which is provided by LINQ as the basis of a database design. The constructor for this class simply calls the constructor for the parent class. The connection string gives location of the database we are going to use. One of the great things about database connection strings is that they can be used to connect to databases over a network, so that a program can work with a database on a distant server. We are not going to do this; we are going to connect the database to a file held in isolated storage on the phone.

The `DataContext` serves as a connection to a database. You can think of it as a bit like a stream that connects a program to a file. It exposes methods that we can use to manage the database contents.

The connection string describes how the program will connect to the database. In some applications this connection can be to a database server on a distant machine. In this case the database requests are sent to the remote system and the program works with the responses. In the case of the database on a Windows Phone this is a path to a file held in Isolated Storage.

Creating Sample Data

When we created our original Customer Manager program we created a method to create sample data. We are going to do exactly the same thing with our database version. This will populate the database with known data which we can then work with. Note that the method is now called `MakeTestDB`, because we will add further test setup for the other tables when we add them later. We can start with exactly the same sample information:

```
public static void MakeTestDB(string connection)
{
    string[] firstNames = new string[] { "Rob", "Jim", "Joe",
                                         "Nigel", "Sally", "Tim" };
    string[] lastsNames = new string[] { "Smith", "Jones",
                          "Bloggs", "Miles", "Wilkinson", "Brown" };

    SalesDB newDB = new SalesDB(connection);

    if (newDB.DatabaseExists())
    {
        newDB.DeleteDatabase();
    }

    newDB.CreateDatabase();

    foreach (string lastName in lastsNames)
    {
        foreach (string firstname in firstNames)
        {
            //Construct some customer details
            string name = firstname + " " + lastName;
            string address = name + "'s address";
            string bank = name + "'s bank";
            Customer newCustomer = new Customer();
            newCustomer.Name = name;
            newCustomer.Address = address;
            newCustomer.BankDetails = bank;
            newDB.CustomerTable.InsertOnSubmit(newCustomer);
        }
    }

    newDB.SubmitChanges();
}
```

The first thing the method does is make a connection to the database:

```
Customers newDB = new Customers(connection);
```

The variable `newDB` now represents a connection to the database with the given connection string. The method can use this connection to issue commands to manage the database.

The next thing that the method does is check to see if the database already exists. If it does, the database is deleted.

```
if (newDB.DatabaseExists())
{
    newDB.DeleteDatabase();
}
```

An important principle of testing is that a test should always start from exactly the same place each time. If there is already a database in a file in isolated storage it will need to be cleared before the test data is added.

```
newDB.CreateDatabase();
```

The next thing the method does is create a new database which contains a set of empty tables.

Once we have our new database we can create a set of test customer values and add each one to the customers table.

```
foreach (string lastName in lastsNames)
{
    foreach (string firstname in firstNames)
    {
```

```
            //Construct some customer details
            string name = firstname + " " + lastName;
            string address = name + "'s address";
            string bank = name + "'s bank";
            Customer newCustomer = new Customer();
            newCustomer.Name = name;
            newCustomer.Address = address;
            newCustomer.BankDetails = bank;
            newDB.CustomerTable.InsertOnSubmit(newCustomer);
        }
    }
```

These two nested loops work through the same set of first and last names we used last time. The major difference from the previous test data generator is that this time a completed customer is added to the database:

```
newDB.CustomerTable.InsertOnSubmit(newCustomer);
```

The `InsertOnSubmit` method is how we add new entries into a table. We are using the `CustomerTable` property of the new database. If the database contained other tables we could add items to them in the same way. Note that this is typesafe; in other words if we tried to add a different type to the `CustomerTable` the program would not compile.

The final, and perhaps the most important, part of the method is the call that actually commits the changes to the database:

```
newDB.SubmitChanges();
```

This is directly analogous to the `Close` method when using file streams. This is the point at which the changes that we have asked for will be committed to the database. Up until the `SubmitChanges` method is called the system may keep some of the changes in memory to speed things up. This makes good sense if the same data item is being updated repeatedly. Rather than changing the database file each time, the system will use copy in memory and work with that. Only when this method is called will the memory copies be written back to the database file.

If you don't submit your changes there is a good chance that this will leave the database in a mess. Which would be bad.

We now have a database which we can use in our application. We can also create a test database with values we can look at. The database context that we are going to use will live in the `App.xaml.cs` program file along with a reference to the currently active customer.

```
public SalesDB ActiveDB;
public Customer ActiveCustomer;
```

When the program starts running the variable `ActiveDB` is set to the database we are using.

```
ActiveDB = new SalesDB("Data Source=isostore:/Sample.sdf");
```

Note that the file path has a particular format and identifies a file in the isolated storage. We can make use of multiple databases in our program if we wish, each will be held in a different file. This database file format is that of a standard SQL database. A PC based database program could read the tables out of this file and it could be managed by an SQL database editor. The reverse is also true, a Windows Phone application could make use of a database that was prepared on a different computer and loaded into the application.

Binding a ListBox to the result of a LINQ Query

Next we want to see how we can use this database in our application. In the simple Customer Manager program we used a `List` to hold all the customers. We found that the `ListBox` display element can take a list and display it. Now we want to take data out of the database and display this.

We get information out of the database by issuing queries. Previously we saw how the SELECT command could be used to fetch information. What we are now going to do is build a LINQ query to get all the customers out of the database.

```
var customers = from Customer customer
                in thisApp.ActiveDB.CustomerTable
                select customer;
```

This line of C# creates a variable called `customers`. The `customer` variable is of type `var`. If you've not seen this type before it can look a bit confusing. What `var` means in this context is "Get the type of this variable from the expression that is being put into it". This doesn't mean that the C# is giving up on strong typing, if we ever try to use the variable `customers` in an invalid way our program will still fail to compile.

The rest of the statement tells LINQ to get all the customer items from the `CustomerTable` member of the `ActiveDB` database context. This is returned as a list of customers that can behave as an `ObserableCollection`. Which is just what we need to give the `ListBox` to display the customers:

```
customerList.ItemsSource = customers;
```

At this point we have a fully working customer database. The only thing that we have to add to the program is code to submit the changes to the database when the user leaves the program. The best way to arrange this is to put the call into the `OnNavigatedFrom` method in `MainPage.xaml.cs`:

```
protected override void OnNavigatedFrom(
                System.Windows.Navigation.NavigationEventArgs e)
{
    App thisApp = Application.Current as App;

    thisApp.ActiveDB.SubmitChanges();
}
```

When the user moves away from this page the method will find the currently active database and submit any outstanding changes to it.

The solution in *Demo 01 SalesManagement* contains a Windows Phone Silverlight application that implements a fully working customer manager. It does persist the data into isolated storage, but because the data storage is created fresh each time you will not see this. Once you have run the program once to create you can comment out the statement in App.xaml.cs that creates a new database and see this working for yourself.

Database and Data Binding Magic

I think this is actually quite magical. We have only added a small amount of code to our application and how we have the data elements stored and retrieved for us automatically. Changes that we make in the user interface are automatically being persisted in the database without any effort from us. It is worth just going through exactly what is happening here so that we can completely understand how it works. Consider the following sequence:

1. Users changes **"Rob Miles"** to "`Rob Miles the Wonderful`" in the `TextBox` holding the name field on the `CustomerDetailsPage`.
2. Because this `TextBox` is bound to the `Name` property in the `CustomerView` bound to this page it now updates that property in the `CustomerView` to the new name text.
3. User presses the `Save` button on the `CustomerView` page.
4. The `saveButton_Click` method in the `CustomerDetailsPage` runs and calls the `Save` method in the `CustomerView` instance behind this page.
5. The `Save` method copies the page properties from the view class into the active customer record.
6. Changes to the `Name` property in the active customer fires the property changed events in LINQ informing the database that the values are being changed.

7. LINQ flags the customer information as having changed and also raises a change event in the `ObservableCollection` which holds our customer list and is connected to the `ListBox` on MainPage.
8. The program returns to the `ListBox` view on `MainPage` and the display of the name "`Rob Miles`" is updated to "`Rob Miles the Wonderful`".
9. When the user exits the application the `OnNavigatedFrom` event fires and calls `SubmitChanges` on the database, storing the changes back into isolated storage.

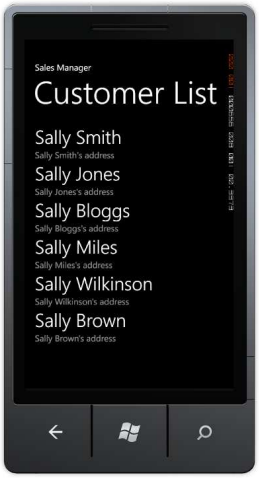
If you don't believe this works, then try it and find out. You can use the above pattern anywhere you want to automatically save and load data with very little effort.

Adding Filters

By slightly changing the format of the LINQ query to fetch the data from the database we can select records that match a particular criteria:

```
var customers = from Customer customer
                in thisApp.ActiveDB.CustomerTable
                where customer.Name.StartsWith("S")
                select customer;
```

This query selects only the customers whose name starts with the letter S.



This would be the result of such a query. This makes it very easy to extract particular elements from the database. We could easily add a search box to our application if we wanted to search for customers with a particular name.

6.2 Creating Data Relationships with LINQ

We are now able to store a large number of customer items and change and update them by using data binding.

However, to make our sales management system work we will need to create other tables and also link them together. To refresh our memories this is the design for the customer table:

Customer ID	Name	Address	Bank Details
123456	Rob	18 Pussycat Mews	Nut East Bank
654322	Jim	10 Motor Drive	Big Fall Bank
111111	Ethel	4 Funny Address	Strange bank

We have created a class which contains the required rows and can use LINQ to persist this data in a database file in isolated storage. The **Products** table is very similar to the

Customers table, in that it just holds a list of items with certain properties.

Product ID	Product Name	Supplier	Price
1001	Windows Phone 7	Microsoft	200
1002	Cheese grater	Cheese Industries	2
1003	Boat hook	John's Dockyard	20

The Products class is actually very similar to the Customer one; it just contains a set of data items. The **Order** table is the more complex one. It actually contains references to rows in the other two classes.

Order ID	Customer ID	Product ID	Quantity	Order Date	Status
1	123456	1001	1	21/10/2010	Shipped
2	111111	1002	5	10/10/2010	Shipped
3	654322	1003	4	1/09/2010	On order

Each row of the Order table describes a particular order that has been placed. It gives the Product ID of the product that was bought, and the Customer ID of the customer that bought it. In C# this would be easy to create:

```
public class Order
{
    public DateTime OrderDate;
    public int Quantity;
    public Customer OrderCustomer;
    public Product OrderProduct;
}
```

The `Customer` and `Order` references would connect an order to the customer that made the order, and the product that was ordered.

However, when we are using a database we don't have any references as such. Rather than having a reference to a customer and product, instead the Order table holds the ID of the items that are related to each order. We then use the value in the Customer ID column to find the customer who has placed the order. For example, we know that order 1 was placed by Rob Miles because it contains a Customer ID of 123456, and so on.

In the good old days before computers this was how things worked. A company would have a filing cabinet full of customer data, another full of product data and a third full of orders. To find out the details of a particular order a clerk would have to look up the customer and product information from information written on the order details. The relational database provided a way for computers to manage information that contains relationships like these, what we want to do now is take the database relationship information and use it to create object references that make it easy to manipulate in an object oriented program.

LINQ Associations

In LINQ a relationship between one table and another is called an Association. This is implemented in the database classes by an `EntityRef` value.

Linking an Order to the Customer that placed it

From our system requirements the `Order` class needs to hold a reference to the `Customer` who placed the order. This is implemented as a relationship between two tables. The `EntityRef` is a special LINQ class we can use to connect two tables together, using a database to provide the underlying storage.

```
[Table]
public class Order : INotifyPropertyChanged,
                     INotifyPropertyChanging
{

    ...

    private EntityRef<Customer> orderCustomer;

    [Association(IsForeignKey = true, Storage = "orderCustomer")]
    public Customer OrderCustomer
    {
        get
        {
            return orderCustomer.Entity;
        }
        set
        {
            NotifyPropertyChanging("OrderCustomer");
            orderCustomer.Entity = value;
            NotifyPropertyChanged("OrderCustomer");
        }
    }
}
```

The `EntityRef` acts as a kind of glue between a simple reference (which we would like to use) and a lookup in a table (which is what LINQ will have to do when we use this reference). The good news is that as programmers we can write:

```
Customer newCustomer = new Customer();
Order newOrder = new Order();
newOrder.OrderCustomer = newCustomer;
```

This simple assignment will have the effect of connecting the order to the customer, so that the database record for that order holds the id of the customer the order is for. We can use this technique every time we want to implement a relationship between a row in one table and a row in another.

When we define the association property we also tell LINQ two other things. We tell it that the association is a “foreign key” and we also identify the property of the class that is going to store the value. If you have used databases before you will be familiar with the idea of a foreign key. This is a primary key from another database. In this case it is the primary key from the Customer database that can be used to identify the particular customer who placed the order.

The storage item in the association tells LINQ which private property of our database will actually hold the property information.

Databases and Navigability

Navigability in database systems is important. It allows a system to find one piece of information given another. At the moment we can start from an order and directly find the customer for that order. This is because a row in the order table contains the `CustomerID` of the customer for that order. However, we can’t find our way back. Starting with an order we have no way of finding out the customer who placed it.

To allow proper navigability we have to put something in the `Customer` class that will allow us to find the orders that the customer has placed. However, things now get even more complicated because a customer could create lots of orders. This means that the nature of the relationships changes depending on which direction you are travelling:

- A particular order will only ever be associated with a single customer, the customer who created the order. This means that the relationship travelling from order to customer is one to one (one order to one customer).

- A particular customer will be associated with many orders. This means that the relationship travelling from customer to order is one to many (one customer to many orders)

When we design our databases we need to consider the navigability, in particular we need to think about how we are going to navigate using our relationships, and whether they are one to one or one to many.

Linking a Customer to all their orders

A customer may place many orders and so the relationship that we need to implement is a “one to many” relationship. If we were implementing this using C# classes we would use a collection of some kind, perhaps an array or a List. In LINQ such a relationship is provided by the `EntitySet` class. This is a bit like the `EntityRef`, except that it can manage a set of items, rather than just one. We can link a customer to their orders by adding an `EntitySet` to the `Customer` class.

```
[Table]
public class Customer : INotifyPropertyChanged,
                    INotifyPropertyChanging
{

    [Column(IsPrimaryKey = true, IsDbGenerated = true,
                                 AutoSync = AutoSync.OnInsert)]
    public int CustomerID { get; set; }


    private EntitySet<Order> orders = new EntitySet<Order>();

    [Association(Storage="orders", ThisKey="CustomerID",
                                   OtherKey="OrderCustomerID")]
    public EntitySet<Order> Orders
    {
        get
        {
            return orders;
        }
        set
        {
            orders = value;
        }
    }
}
```

This looks very like the `EntityRef` that we created earlier, except that it has extra information in the Association to describe the relationship.

The item `ThisKey` gives the name of the property that will be used by the `Order` to find the customer it is associated with. We can use `CustomerID`, the primary key for the `Customer` to do this.

The item `OtherKey` gives the name of the element in the `Order` class that will implement the association in the other direction, i.e. allows an order to find the customer it is related to.

You may be wondering why we have not fired any `NotifyPropertyChanged` events when `Orders` is changed. This is because changing the value of the `EntitySet` is something that we would hardly ever do. Remember that this is the container for the orders, not the orders themselves. We will add orders to the customer by putting things into the `EntitySet`, not by replacing the `EntitySet` itself.

To make the association work properly we have to make some changes to the reference in the `Order` class. These will bind both ends of the association together.

```
[Table]
public class Order : INotifyPropertyChanged,INotifyPropertyChanging
{

    ...

    [Column]
    public int OrderCustomerID;

    private EntityRef<Customer> orderCustomer =
                                        new EntityRef<Customer>();

    [Association(IsForeignKey = true, Storage = "orderCustomer",
                                         ThisKey = "OrderCustomerID")]
    public Customer OrderCustomer
    {
        get
        {
            return orderCustomer.Entity;
        }
        set
        {
            NotifyPropertyChanging("OrderCustomer");
            orderCustomer.Entity = value;
            NotifyPropertyChanged("OrderCustomer");
            if (value != null)
                OrderCustomerID = value.CustomerID;
        }
    }
}
```

The Association is defined as holding a foreign key (the primary key of the customer database) and using the value of `OrderCustomerID` to hold this value.

This version of the reference makes a copy of the `CustomerID` value into the `OrderCustomerID` column in the table when a customer is assigned to the order. In other words, if I write something like this:

```
Customer c = new Customer();
Order o = new Order();
o.OrderCustomer = c;
```

When the `OrderCustomer` property is assigned the set method above will take the `CustomerID` value and copy it into the `OrderCustomerID` value to record the customer for this order.

If all this is hurting your head a bit, and I must admit it hurt mine first time round, remember the problem that we are trying to solve. The `ThisKey` and the `OtherKey` descriptions in the association tell each side of the relationship how to find each other.

The good news is that once we have implemented this pattern we can easily add new orders to a customer:

```
Customer c = new Customer();
Order o = new Order();
o.OrderCustomer = c;
c.Orders.Add(o);
```

The `EntitySet` class provides an `Add` method that lets us add orders to the database. To work through the contents of the entity set we can use any of the constructions we would normally use to process collections:

```
foreach (Order o in c.Orders)
{
    // do something with each order
}
```

This doesn't look that special, but the special thing is that we are writing C# to work with the database.

Orders, Products and Database Design

We now have a link between customers and orders which works property. A customer can have a very large number of orders, and an order is always connected to the customer who placed it. The next thing we need to do is fill in the order with the products that the customer might have bought. This is the point at which we hit a snag with our original design of our `Order` table:

Order ID	Customer ID	Product ID	Quantity	Order Date	Status
1	123456	1001	1	21/10/2010	Shipped
2	111111	1002	5	10/10/2010	Shipped
3	654322	1003	4	1/09/2010	On order

If you look carefully at the design it turns out that there is no way we can have an order that holds more than one product. A particular order can be for multiple items, but they must all be of the same item.

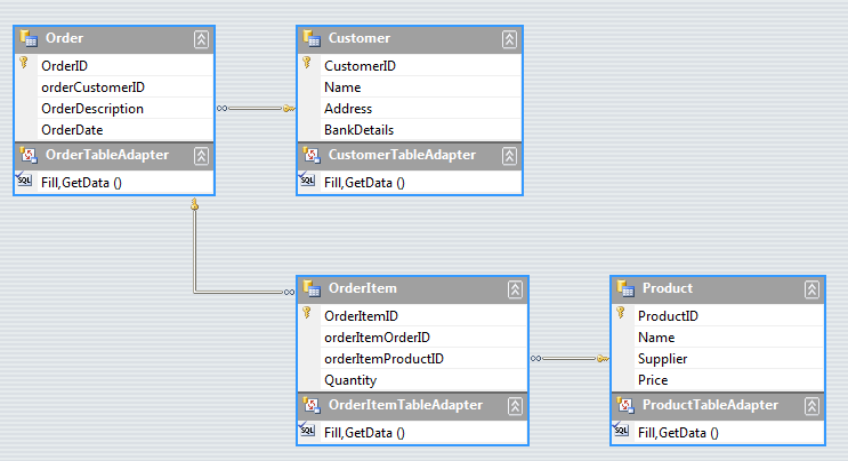
It might be that this is perfectly OK. The person who wants to use your system might be selling items that are very expensive and are only ever ordered one at a time. Alternatively this might be disastrous; the system may be used by a supermarket that wishes to assemble orders which contain a huge number of different products.

To solve the problem we need to add another table, which we will hold all the items in each order. Each `OrderItem` will be linked to a particular order. Each `Order` will then hold a collection of these items.

OrderItem ID	Order ID	Product ID	Quantity
1	56	1001	1
2	56	1002	5
3	12343	1003	4

The table above shows that the order 65 was made up of two products, a Windows Phone and 5 Cheese Graters. (If you are wondering where the products come from, take a look at the Products table way back at the start of this chapter).

Database designers like to use diagrams to show how the various elements fit together. The diagram for the database that we have built looks like this:



The little keys show where a foreign key is being used at one end of the association. The little infinities (∞) show that an association can connect to any number of that item.

If you do a lot of database design you will be used to drawing diagrams like the one above. It was actually created from the database that was formed by the classes that we created.

The solution in *Demo 02 SalesManagement* contains a Windows Phone Silverlight application that adds Order, OrderItem and Product tables to the customer database. You can navigate the lists of customers and products and also find the orders that a customer has created and their contents.

LINQ Queries and Joining

Now that we have our data in a database we can use the very powerful query mechanisms to get information from it. For example, we might want to find all the orders that were placed on a particular date.

```
DateTime searchDate = new DateTime(2011, 8, 19);

var orders = from Order order
                in activeDB.OrderTable
                where order.OrderDate == searchDate
                select order;
```

This query searches the order table of the currently active DB and finds all the orders created on the 19th of August 2011. We don't have to write any of the searching; the result is just returned in the variable called `orders`. You might be wondering what the actual type of `orders` is. Actually it is of type "LINQ query". The statement above doesn't do anything until you try to consume the data that it holds. For example, if we tried to work out the total value of the sales for that day we could write something like this:

```
int totalSales = 0;

foreach (Order order in orders)
{
    foreach (OrderItem item in order.OrderItems)
    {
        totalSales += item.OrderItemProduct.Price;
    }
}
```

The first `foreach` works through `orders` getting each order in turn. It is only when the program starts to actually consume the data that LINQ will leap into action and start producing results.

Note that this has implications if you want to use the result of a query more than once. If you use the same query twice you will find that your program will run a bit more slowly, as LINQ will be fetching things from the database twice. If you want to use a

result more than once it is best to use the query to product a list of items that you can then work through as many times as you like:

```
List<Order> DateOrders = orders.ToList<Order>();
```

This creates a list called `DateOrders` which contains the result of the query. We can use the list in our program without generating any further transactions on the database. The `ToList` method applies the query and generates a list of results.

Joining two tables with a query

LINQ has another trick up its sleeve. It can create queries that go across more than one table, a process called "joining". Perhaps we might want to create a complete list of all the orders made by all customers, listed in customer order. We want to find `Order` and `Customer` pairs that match up and list them. Using the `join` keyword LINQ can do this in one action:

```
var allOrders = from Customer c in newDB.CustomerTable
        join Order o in newDB.OrderTable on
                            c.CustomerID equals o.OrderCustomerID
        select new { c.Name, o.OrderDescription };
```

If you haven't seen one of these you would be forgiven wondering how on earth it compiles. The key is in the name LINQ, in that the `from`, `join` and `select` keywords are all integrated into the C# language to make these kinds of queries work. The first part of query finds all the customers in the customer table. This query is then joined onto a second, which works through all the orders doing a join on the two ID values stored in customer and order.

Essentially what it is saying is "Find me pairs of orders and customers where the orderID in the customer matches the customer id in the order".

Once it finds a match it then creates a new type that just contains the results that we want, in this case the `Name` from the customer and the `OrderDescription` from the order. Note the magic `var` is here again. In this case we get an SQL query that generates a collection of type containing two strings of the appropriate names.

We could use this query to produce a list of order descriptions by working through it

```
List<String> OrderDescriptions = new List<String>();

foreach (var orderDesc in allOrders)
{
    OrderDescriptions.Add(orderDesc.Name + " order: " +
                          orderDesc.OrderDescription);
}
```

As with the previous query, LINQ doesn't actually do anything until the program starts to work through the results. Note that we need to use a `var` type for the `orderDesc` value that we are fetching from the query. This contains the two items that we added when we created the type in the query.

The result of his query could be bound to a display element to show the list:

```
Rob Miles order: Camera Order
Rob Miles order: Cheese Order
Rob Miles order: Barge Pole Order
```

The var keyword

The `var` keyword can seem a little scary if, like me, you are used to declaring variables of a particular type and then just using them. I like the way that the C# compiler fusses about my programs and stops me from doing stupid things like:

```
int i = "hello";
```

An attempt to put a string into an integer is a bad thing to do, and will not compile. However, at first glance the `var` keyword looks like a way to break this lovely safety net. We can create things that don't seem to have a type associated with them, which must be bad. However, you can rest assured that although we don't give a name for a type like this, the C# compiler knows exactly what a `var` type is made from, and will refuse to compile a program that tries to use such a type in an incorrect way.

Deleting Items from a Database

We have seen how to add items to a database. It is also possible to delete items as well. For example, if the customer decided that they didn't want a particular order item we can delete it from the database:

```
ActiveDB.OrderItemTable.DeleteOnSubmit(item);
```

We can also delete collections of items using `DeleteAllOnSubmit`, including a collection identified as a result of a query.

Foreign Key Constraints and Delete

Some of our data items have relationships with others, for example a Customer will contain a number of Orders, and an Order will contain a number of OrderItem values. If the parent object is deleted before the children (i.e. if we try to delete a customer which contains some orders) the database will refuse to perform this, and will throw an exception as a result. If you want to delete a customer you will first have to delete all the order items in each order, and then delete the orders, and then delete the customer.

What We Have Learned

1. Windows Phone applications can use Language Integrated Query (LINQ) to interact with databases can be stored in files in isolated storage in the device. The databases themselves are managed by an SQL database server, but it is not possible to use SQL commands to interact with a database, instead queries are expressed using LINQ.
2. A database is made up of a number of tables. A table is made up of columns (different data items such as name, address, and bank details) and rows (all the information about a particular item – the name, address and bank details of an individual customer).
3. One column in a table acts as the "primary key", holding a value which can uniquely identify that row (the unique ID of that particular customer). The database can be asked to automatically create primary keys that are unique for each row in a table.
4. Windows Phone applications can create classes which are stored by LINQ as database tables. The attributes [Table] and [Column] are used within the class definition to specify the role of the components. Additional information can be added to the attributes to identify primary keys.
5. Database classes managed by LINQ can add notification behaviours to the data properties so that LINQ will automatically update the database entries when the properties are changed. If these are used with changed events in Silverlight display elements this can allow automatic database update when items are changed by the user.
6. A database can contain relationships, where a column in one table contains primary key values which identify rows in another. These are called "foreign keys" as they are primary keys, but not for the table which contains them.
7. LINQ implements relationships by using the EntityRef class, which contains a reference to a foreign key in another database. This is used for "one to one" relationships. The EntitySet class is used to allow a member of a table to refer

to a large number of rows in another table. This is used for “one to many” relationships. These data elements are modified with the Association attribute that describes how the table relationship is implemented.

8. A database will frequently have a one to one going in one direction (an order is attached to one customer) and a one to many going in the other (a customer can place many orders). For such associations to work correctly is important that the associations are correctly configured and that foreign keys are stored correctly.
9. A LINQ query can be used to fetch structured data from the database. The query is only actually executed when the results of the query are being consumed by the program. LINQ queries can use the join keyword to combine the results of multiple queries.
10. When deleting items from a database all child items must be deleted before the parent object is removed, otherwise there would be elements in the database containing primary key values for items that don’t exist.