

```

public class LectDataSet : DataSet
{
    public LecturersDataTable Lecturers;
    public OfficesDataTable Offices;
    ...
}

public class LecturersDataTable : DataTable, IEnumerable
{
    public LecturersRow this[int index] { get {...} }
    public LecturersRow FindByLectID(int lectID) { ... }
}

public class OfficesDataTable : DataTable, IEnumerable
{
    public OfficesRow this[int index] { get {...} }
    public OfficesRow FindByOfficeID(int officeID) { ... }
}
    
```

```

public class LecturersRow : DataRow
{
    public int LectID { get {...} set {...} }
    public string Name { get {...} set {...} }
    ...
    public OfficesRow[] GetOfficesRows() {...}
}

public class OfficesRow : DataRow
{
    public string Building { get {...} set {...} }
    public string Room { get {...} set {...} }
    ...
    public LecturersRow LecturersRow
        { get {...} set {...} }
}
    
```

```

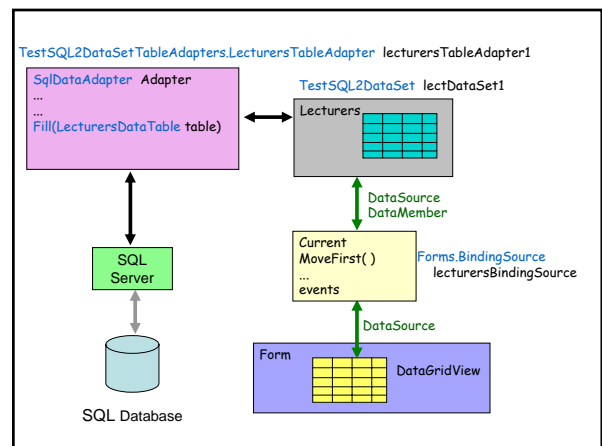
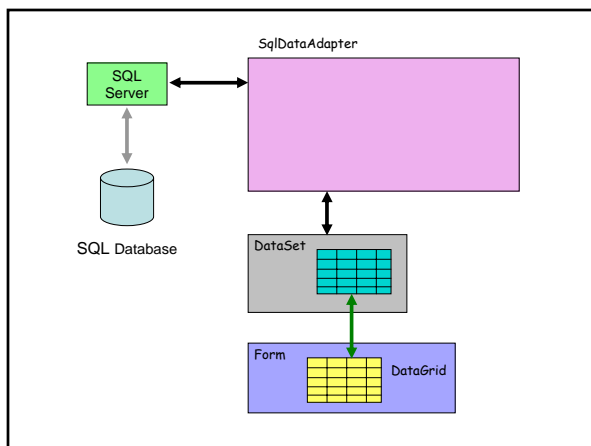
namespace LectDataSetTableAdapters
{
    public partial class LecturersTableAdapter
    {
        ...
    }

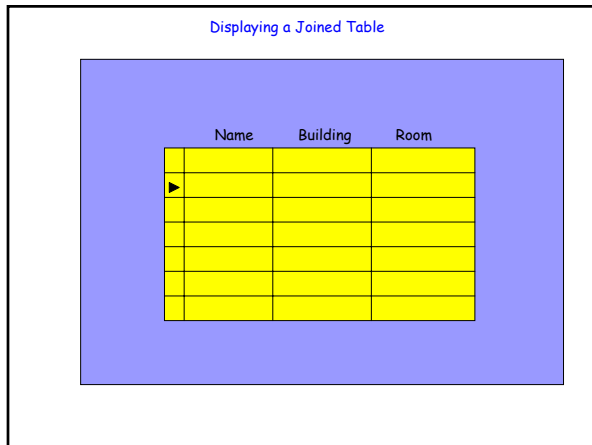
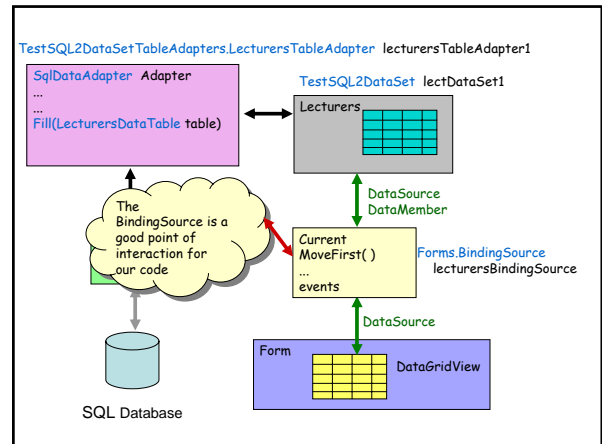
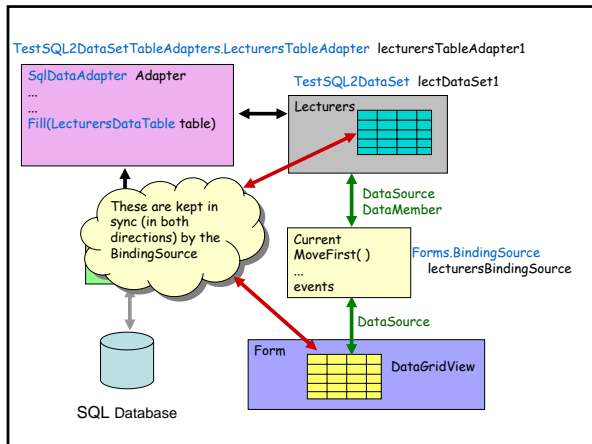
    public partial class OfficesTableAdapter
    {
        ...
    }
}
    
```

```

namespace LectDataSetTableAdapters
{
    public partial class LecturersTableAdapter
        : System.ComponentModel.Component
    {
        ...
    }

    public partial class OfficesTableAdapter
        : System.ComponentModel.Component
    {
        ...
    }
}
    
```





Database Operations

- SELECT - retrieve rows from a table
- INSERT - add new rows to a table
- UPDATE - modify the data in existing rows
- DELETE - remove rows from a table

How do we do these operations

- to the dataset?
- to the database?

Working with DataSets

- DataSets are in-memory copies of the data
- The Fill method uses a SELECT command to retrieve data
- We can modify data directly—easy with typed DataSets with methods like **AddOfficesRow** and **RemoveOfficesRow** and we have easy access to the data through tables and their rows
- DataSets have **Add** and **Delete** methods exported by their DataRowCollection objects

Working with DataSets

- Rows have a **RowState** property whose value comes from the **DataRowState** enumeration

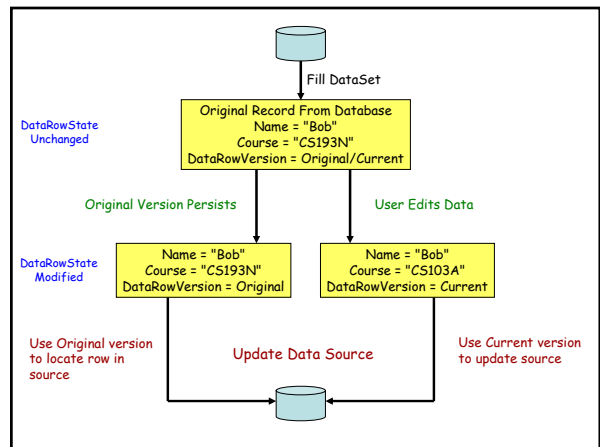
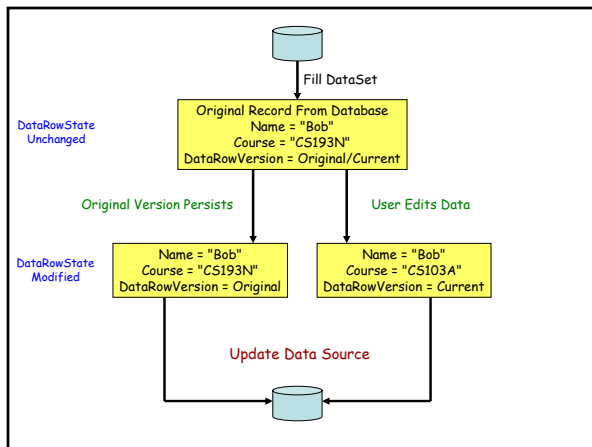
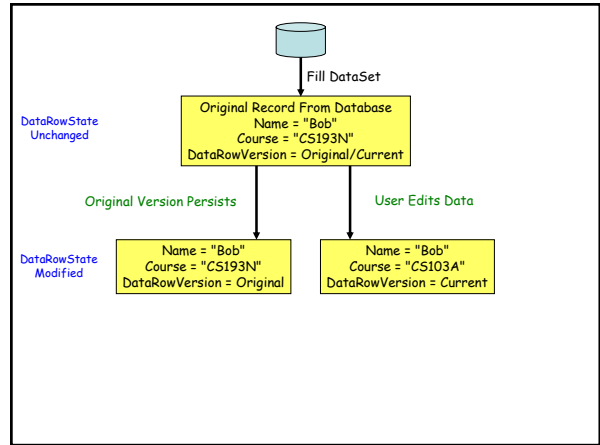
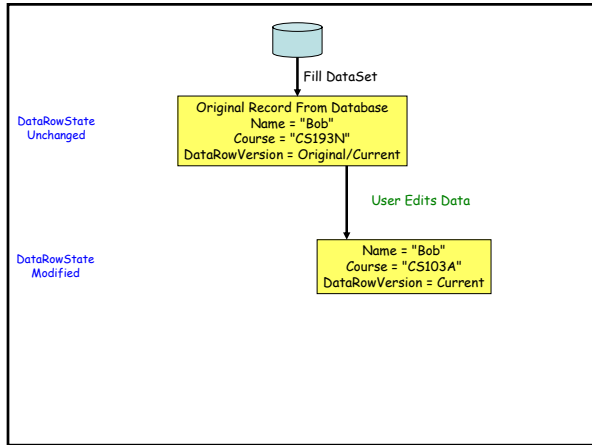
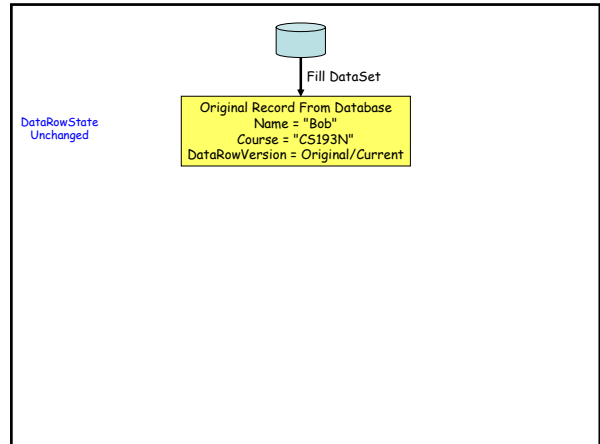
```

Added
Deleted
Detached
Modified
Unchanged
    
```

row.RowState

Working with DataSets

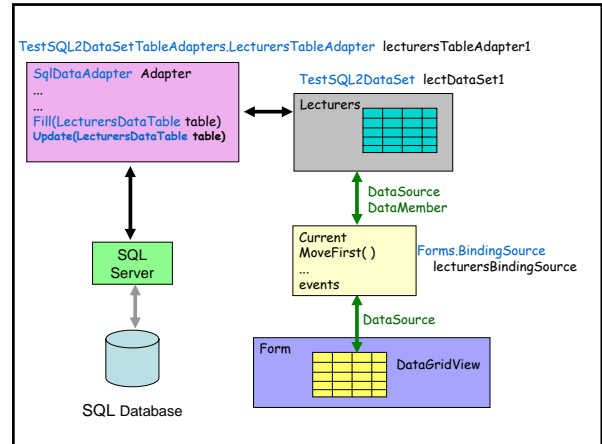
- Rows have a **RowState** property whose value comes from the **DataRowState** enumeration
  - Added
  - Deleted
  - Detached
  - Modified
  - Unchanged
- Rows maintain multiple versions, accessible using a member of the **DataRowVersion** enumeration
  - Current
  - Original `row[3, DataRowVersion.Original]`
  - Proposed
  - Default



We can accomplish the update by using the TableAdapter's Update method:

```
officesTableAdapter.Update(testSQL2DataSet.Offices);
```

The underlying SqlDataAdapter examines the RowState property for each row, and executes the required INSERT, UPDATE, or DELETE statement. The state of all rows in the DataSet is then set to Unchanged.



### Database Updating

In a multiuser environment, suppose:

- Two users have copies of the same data
- Both change it
- One user updates the database
- Then the other user updates the database

What happens?

### Database Updating

There are three models for updating the data in the database:

- Pessimistic concurrency
- Optimistic concurrency
- "Last in wins"

### Database Updating

**Pessimistic concurrency**—rows are locked at the data source until the lock owner releases it

Useful where

- There is high contention for the same records. The cost of placing locks may be less than the cost of dealing with conflicts.
- The nature of the user's task requires that data not be changed while the steps are carried out.

Difficult in a disconnected architecture

Not good where lock times are long and there are many users

### Database Updating

**Optimistic concurrency**—rows are not locked when they are read. A concurrency error occurs if a user attempts to modify the database but finds that the data has been changed.

## Database Updating

Optimistic concurrency—rows are not locked when they are read.  
A concurrency error occurs if a user attempts to modify the database but finds that the data has been changed.

## Useful where

- There is low contention for the same records.
- High performance is desired
- Code can be written to handle errors

A good fit for a disconnected architecture

## Database Updating

Last in Wins—no check of the original data is made.  
The data is always written to the database.

User A reads a row.

User B reads the same row, modifies it, and writes it to the database.

User A modifies the row and writes it to the database.

B's changes are never seen by A and are overwritten.

## Database Updating

SQL Update commands:

```
UPDATE R SET <new value assignments> WHERE <condition>
```

## Database Updating

SQL Update commands:

```
UPDATE R SET <new value assignments> WHERE <condition>
```

```
UPDATE Lecturers SET Course = 'CS106A'
WHERE LectID = '1'
```

## Database Updating

SQL Update commands:

```
UPDATE R SET <new value assignments> WHERE <condition>
```

```
UPDATE Lecturers SET Course = 'CS106A'
WHERE LectID = '1'
```

What we put in the WHERE clause determines what happens.

## Database Updating

```
UPDATE R SET <new value assignments> WHERE <condition>
```

Include only the primary key

```
UPDATE Lecturers SET Course = 'CS106A'
WHERE LectID = '1'
```

Last in wins

**Database Updating**

UPDATE R SET <new value assignments> WHERE <condition>

Include all columns

```
UPDATE Lecturers SET Course = 'CS106A'
WHERE LectID = '1' AND Name = 'Bob' AND
Course = 'CS193N' AND Students = '99'
```

Optimistic concurrency

**Database Updating**

UPDATE R SET <new value assignments> WHERE <condition>

Include all columns

```
UPDATE Lecturers SET Course = 'CS106A'
WHERE LectID = '1' AND Name = 'Bob' AND
Course = 'CS193N' AND Students = '99'
```

Optimistic concurrency

If another user has changed the data, no rows will match, and a Concurrency Exception will occur

**Database Updating**

UPDATE R SET <new value assignments> WHERE <condition>

Include primary key and a timestamp column

```
UPDATE Lecturers SET Course = 'CS106A'
WHERE LectID = '1' AND
Timestamp = 'original value'
```

Optimistic concurrency

If another user has changed the data, no rows will match, and a Concurrency Exception will occur

**Database Updating**

UPDATE R SET <new value assignments> WHERE <condition>

Include primary key and modified columns

```
UPDATE Lecturers SET Course = 'CS106A'
WHERE LectID = '1' AND
Course = 'CS193N'
```

Succeeds if another user has changed another column on this row.

**Database Updating**

UPDATE R SET <new value assignments> WHERE <condition>

Include all columns

```
UPDATE Lecturers SET Course = 'CS106A'
WHERE LectID = '1' AND Name = 'Bob' AND
Course = 'CS193N' AND Students = '99'
```

Optimistic concurrency

If another user has changed the data, no rows will match, and a Concurrency Exception will occur

Visual Studio does by default.