

Chapter Overview



- **Objectives**
 - Explain the ADO object hierarchy
 - Access a database using ADO
- **Chapter content**
 - Architecture - ADO and OLEDB
 - Connection objects - connecting to data sources
 - Command objects - executing commands
 - Recordset objects - retrieving data
 - Disconnected recordsets - marshaling by value
- **Practical content**
 - Use ADO to retrieve information from a database
- **Summary**

352

What is ADO?

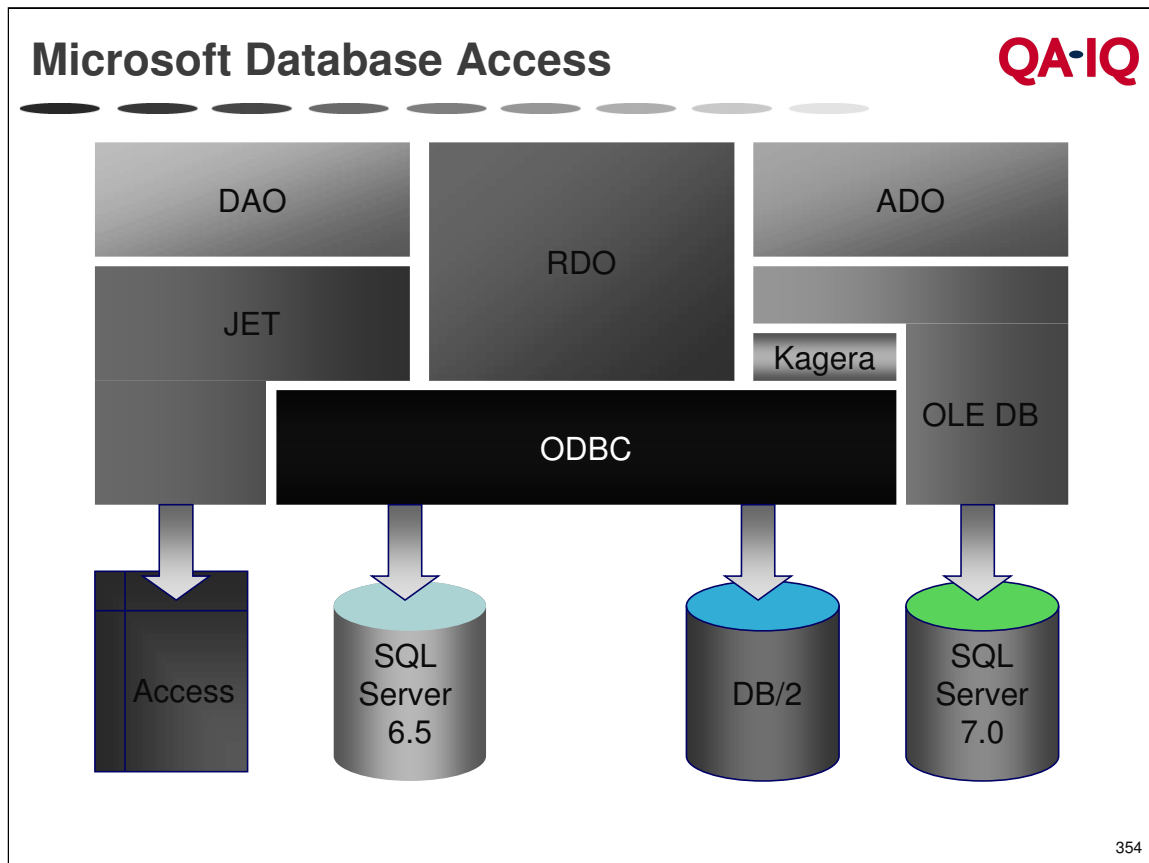


- **Active Data Objects**
- **Set of COM objects**
 - **Command, Connection, Recordset, Field, Error**
 - **dual interfaces**
- **A specification**
 - **ADODB**
 - **ADOR**
- **An OLE DB consumer**
 - **Talks to OLE DB interfaces**

353

ADO stands for ActiveX Data Objects, a specification for an Automation wrapper around the Universal Data Access interfaces, also known as OLE DB interfaces. ADO provides several objects, the main ones being Connection, Command and Recordset. Each object has it's own methods, properties and collections which are exposed through dual interfaces, callable through `IDispatch::Invoke` or v-table offset.

Because ADO is a specification there may be several implementations. ADODB is the standard implementation, ADOR implements just the Recordset object and it's associated collections. ADOfx was another implementation available in the early days of OLE DB. It has subsequently become the Microsoft Client Cursor Engine.



There are now many different ways to access data. Microsoft provide several automation wrappers around several different data access technologies. DAO (Data Access Objects) is a wrapper around JET (Joint Engine Technology) a data access technology designed for manipulating flat-file database formats such as Microsoft Access. JET can also access other data sources via ODBC. RDO (Remote Data Objects) is a wrapper on top of the ODBC (Open DataBase Connectivity) API. ADO (ActiveX Data Objects) is a wrapper around OLE DB. OLE DB is a generic COM based data access layer that is not restricted to relational data. OLE DB can access ODBC data sources through the OLEDB-ODBC bridge, known as Kagera.

Each of the wrapper technologies (DAO, RDO and ADO) are hierarchies of COM objects, representing underlying database concepts such as connections, commands and resultsets. Of these 3 wrappers, ADO is the newest, and is also Microsoft's preferred technology for data access. It is also the easiest to learn from scratch as it has the simplest object model.

Database access in MTS is not restricted to ADO. RDO can be used, as can the ODBC API.

ADO Implementations



- **ADODB**
 - **Server Cursors**
 - **Optimised for use with Microsoft OLE DB providers**
 - **All cursors available**
- **ADOR**
 - **Client Cursors**
 - **Recordset only**
 - **Used for creating disconnected, client-side Recordsets**
 - **Initially designed for HTTP scenarios with Remote Data Services**
 - But also useable over DCOM
 - **Forward-only and Static cursors**

355

There are several implementations of ADO. As ADO is a specification, in theory anyone could provide an ADO implementation, either a generic one or one optimized for a particular task.

ADODB is a generic implementation, providing each object in the ADO object model. ADODB is optimized for use with Microsoft OLE DB providers, including the OLEDB-ODBC bridge. ADO DB 1.0 was the first version which shipped with the OLE DB SDK 1.0 and IIS 3.0.

ADOR is an implementation of only the Recordset object.

ADO DB 1.0 included a fully functional Recordset object and a cut down Recordset was provided with ADOR 1.0. With the advent of ADO 1.5, ADO DB no longer contains a Recordset object. The Recordset object is provided by ADOR 1.5.

The reason for having a separate Recordset object is so that it can be installed separately from ADO DB. Many client applications may only require Recordset functionality and not need the other objects in the ADO object model. This also allows for the creation of disconnected recordsets that are located on the client side. The client can then move through the recordset without having to repeatedly request new records from the server. This reduces network round trips. This technology was originally designed for use with Remote Data Services (nee Advanced Data Connector) but also works across DCOM.

In either case the recordset is marshaled into the client address space, so the client has a copy of the records locally. The recordset can be updated by the client and then subsequently sent back to the server.

ADO Advantages



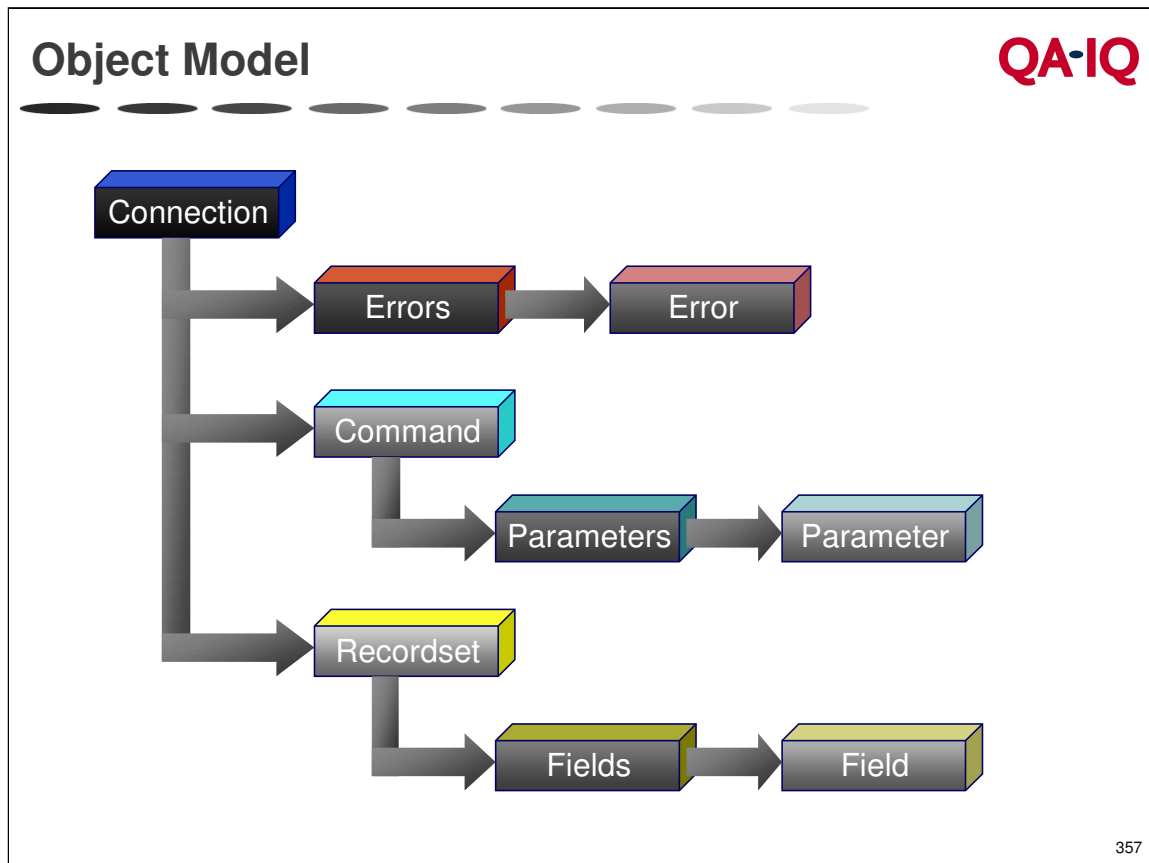
- **Not hierarchical**
 - **Can create objects without their 'parent'**
- **Very flexible**
 - **Collections for**
 - Query parameters
 - Rowset fields
 - Dynamic properties
- **Free threaded**
 - **Performance gain**
 - **BUT requires thread safe ODBC driver if using Kagera**

356

ADO has many advantages over DAO, RDO or the ODBC API. It is not a strict hierarchy of objects as DAO and RDO are. Objects can be created independently of their 'parent'. For example a Recordset or Command object can be created without first creating a Connection object.

ADO is very flexible. Recordsets are self-describing, Command and Connection objects can provide custom information about the underlying OLE DB provider through a dynamic properties collection. Commands support parameterised queries, with both input and output parameters, through a Parameters collection.

ADO also supports the COM free threading model which will provide better performance than apartment model objects, especially with COM+ supporting the 'neutral' threading model. In order to use ADO free threaded with Kagera the underlying ODBC driver must be thread safe. This is also true for MTS (in addition the ODBC driver must have no thread affinity), so it should not be an issue when designing MTS applications. As the ODBC driver will need to support thread safety for MTS, ADO will be able to run free threaded.



The diagram shows the ADO object model. The Connection object has an Errors object. The Errors object is a collection of Error objects. A Command object will be associated with a Connection object, implicitly or explicitly. The Command object has a Parameters object, which is a collection of Parameter objects. The Parameters collection is used to support parameterised queries.

A Recordset object may also be associated with a Connection object, again implicitly or explicitly. A Recordset has a Fields object, which represents a row of data in relational database terms. The Fields object is a collection of Field objects, each Field object representing a data item in the current row in the database (when talking to a relational system).

Connection, Command, Recordset and Field objects also have a Properties object which is a collection of Property objects. Properties are named values and are object dependent. The properties in the Properties collection are known as dynamic properties. They differ from the standard properties exposed by the various objects in the ADO object model in that they are defined by the underlying OLE DB provider.

ADO Type Library and Smart Pointers



- **Type Library**

```
#import "C:/Program Files/Common
Files/System/ado/msado15.dll"
        rename("EOF", "ADOEOF")
using namespace ADODB;
```

- **Create using ProgID**

```
_ConnectionPtr sConnection("ADODB.Connection");
```

- **Create using GUID**

```
_ConnectionPtr sConnection( __uuidof(Connection) );
```

- **Delayed Creation**

```
_ConnectionPtr sConnection;
sConnection.CreateInstance(__uuidof(Connection) );
```

358

Most ADO code is written in Visual Basic because most developers consider writing ADO code in C++ is too difficult. It is certainly true that writing ADO code in raw C++ can be very problematical, but if you use the smart pointers provided by the ADO type library, using ADO in C++ can be a breeze.

Firstly, you must get access to the ADO type library. Although we now use version 2.5 of the ADO library we access the type library with:

```
#import "C:/Program Files/Common Files/System/ado/msado15.dll"
rename("EOF", "ADOEOF")
using namespace ADODB;
```

The version number hasn't been updated! Note that the type library defines the **ADODB** namespace. It is advisable to rename **EOF** to **ADOEOF** during the type library import to avoid name clashes.

To create ADO objects (only Connection objects are shown here) you have several choices. You can create ADO objects immediately using a ProgID or a GUID at the time you declare your smart pointer:

```
_ConnectionPtr sConnection("ADODB.Connection");
_ConnectionPtr sConnection(__uuidof(Connection));
```

This is convenient for declaring local objects.

Alternatively you can declare a pointer to an ADO object and instantiate it later:

```
_ConnectionPtr sConnection;
sConnection.CreateInstance(__uuidof(Connection));
```

This is convenient for declaring global objects.

Connection Object



Methods

Close(void)

Execute(BSTR *CommandText*, VARIANT **RecordsAffected*, long *Options*,
ADOREcordset ***ppiRset*)

Open(BSTR *ConnectionString*, BSTR *UserID*, BSTR *Password*, long *Options*)

Properties

get_ConnectionString(BSTR **pbstr*)

put_ConnectionString(BSTR *bstr*)

get_ConnectionTimeout(LONG **pTimeout*)

put_ConnectionTimeout(LONG *Timeout*)

Events

ConnectComplete(ADOError **pError*, EventStatusEnum **adStatus*,
_ADOConnection **pConnection*)

Disconnect(EventStatusEnum **adStatus*, _ADOConnection **pConnection*)

359

The Connection object is used to open a connection to a data source, usually a specific database on a specific server. The Connection object can then be used to execute command directly via the `Execute` method. Alternatively the connection object could be associated with a Command or Recordset object.

The `ConnectionString` property can be set explicitly or set by passing a parameter to `Open`. In either case it can be a DSN (Data Source Name), or a detailed connection string, e.g. "DSN=MyData;UID=Client;PWD=Kidney"

The Connection object is also responsible for maintaining an Errors collection for errors that occur on the connection. This Errors collection can be used to retrieve detailed information the last error that occurred. Each error may generate one or more Error objects which are placed in the Errors collection. Note that each error that occurs will clear the Errors collection and then repopulate it with new Error objects.

The Connection object also supports transactions through the `BeginTrans`, `CommitTrans` and `RollbackTrans` methods. These are not of much interest when running under MTS as all transaction management will be provided by MTS.

Using a Connection Object



```

_ConnectionPtr sConnection;

STDMETHODIMP CPhones::SetupConnection()
{
    // Create a connection object
    sConnection.CreateInstance("ADODB.Connection");
    // Set up connection DSN and open connection
    _bstr_t fileDSN = "FileDSN=C:\\MyDSNs\\Phones.dsn";
    sConnection->Open(fileDSN, (BSTR)0, (BSTR)0, -1);
    // Execute a command
    bstr_t SQL("DELETE * FROM Temp WHERE Bought='N'");
    sConnection->Execute(SQL, NULL, adExecuteNoRecords);
    // Close the connection
    sConnection->Close();
    return S_OK;
}

```

360

This slide shows how to use a Connection object to execute an SQL query on a database. Smart pointers are used throughout.

The code how to set up a smart pointer to a Connection object:

```

_ConnectionPtr sConnection;

```

At this stage the pointer sConnection is null; a connection has not yet been established. In the SetupConnection method a connection object is created using a ProgID.

```

sConnection.CreateInstance("ADODB.Connection");

```

Before the connection object can be used, we must open a connection to an existing database. It is very convenient to use a File DSN to establish a connection:

```

_bstr_t fileDSN = "FileDSN=C:\\MyDSNs\\Phones.dsn";
sConnection->Open(fileDSN, (BSTR)0, (BSTR)0, -1);

```

Notice that the Open method must have all 4 parameters supplied, even though the DSN contains all the information required to open the connection. The last 3 parameters are set to the default values (default as far as VB is concerned).

Now we are ready to issue an SQL command to the database:

```

_bstr_t SQL("DELETE * FROM Temp WHERE Bought='N'");
sConnection->Execute(SQL, NULL, adExecuteNoRecords);

```

When we have finished with the connection object it is important to close the connection:

```

sConnection->Close();

```

prior to the object going out of scope.

Command Object



Methods

```
Cancel(void);

CreateParameter(BSTR Name, DataTypeEnum Type, ParameterDirectionEnum
    Direction, long Size, VARIANT Value, _ADOParameter **ppiprm);

Execute(VARIANT *RecordsAffected, VARIANT *Parameters, long Options,
    _ADORecordset **ppirs);
```

Properties

```
get_ActiveConnection(_ADOConnection **ppvObject);
put_ActiveConnection(VARIANT vConn);
putref_ActiveConnection(_ADOConnection *pCon);

get_CommandText(BSTR *pbstr);
put_CommandText(BSTR bstr);
```

361

The Command object is used to define a command to be executed against a data source. As such it must be associated with a connection, either implicitly or explicitly. The `Execute` method is used to run the command against the data source. It returns a Recordset object containing the results of the command.

The `CommandText` property is the actual command to be execute, e.g. "SELECT * FROM PEOPLE"

The `ActiveConnection` property defines the database connection that the command will be executed against. This can be either a reference to an existing Connection object or a connect string. A command can be executed against multiple connections by setting the `ActiveConnection` property and calling `Execute` repeatedly.

The Parameters collection is used to support parameterised queries, either standard SQL statements or stored procedures. Both input and output parameters are supported.

Using a Command Object



```
STDMETHODIMP CPhones::ExecuteCommand()
{
    // Create a command object
    _CommandPtr sCommand("ADODB.Command");

    // Set up connection
    sCommand->ActiveConnection = sConnection;

    // Execute a command
    sCommand->CommandText = "myStoredProcedure";
    sCommand->CommandType = adCmdStoredProc;
    sCommand->Execute(0, 0, adCmdStoredProc, 0);
    return S_OK;
}
```

362

Command objects are used when we do not need to extract a recordset from our database. The command object is created with:

```
_CommandPtr sCommand("ADODB.Command");
```

Command objects still require a connection. To use an existing connection defined as a smart pointer simply write:

```
sCommand->ActiveConnection = sConnection;
```

We can now issue a command such as a stored procedure with:

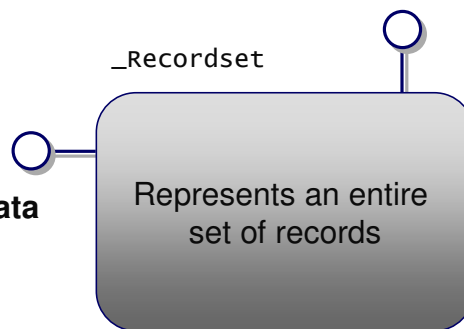
```
sCommand->CommandText = "myStoredProcedure";
sCommand->CommandType = adCmdStoredProc;
sCommand->Execute(0, 0, adCmdStoredProc, 0);
```

More complex commands may be attempted. For example, you can pass parameters to a stored procedure. The on-line help (MSDN July 1999) contains example code on how to do this.

Recordset Object

QA-IQ

- **Open, Close and Requery** (M)
- **MoveFirst, Move Last, MoveNext, MovePrevious and Move** (M)
 - Navigate through rows
- **AddNew and Update** (M)
- **BOF, EOF, RecordCount** (P)
- **Fields** (C)
 - Field objects contain column data
 - Allows retrieving and updating of data



363

The Recordset object provides access to the actual data in the data source. A Recordset represents an entire set of records, often the result of a SQL SELECT statement. The Recordset object provides navigation features and the ability to update the data. A Recordset has a set of Methods (M), Properties (P) and Collections (C), just like any other COM object.

The various MoveXXX methods are used to navigate through the recordset. Note that some MoveXXX methods may not be available with some CursorTypes, e.g. MovePrevious would not be available with a Forward-only cursor.

AddNew creates a new record. This can be empty if both parameters to AddNew are blank. Or the data can be specified when AddNew is called by passing an array of values. If a blank record is created it can be filled in by setting the Value property of the various Field objects in the Fields collection.

Update is used to write changes made to the current record back to the database.

The RecordCount property contains the number of records in the recordset. This is the total number of records returned by the SQL command or Table. Note that for some providers this property is only valid after the cursor has been scrolled to the last record in the recordset. This means that a MoveLast should be executed before relying on the value of RecordCount.

If the recordset is empty, that is has no records, then RecordCount will be set to zero, and a MoveLast is unnecessary. BOF and EOF will also be TRUE in this case.

The Fields collection is a collection of Field objects, each one representing a column in the database. The Field objects can be used to retrieve values from the current row, and to update the values. Various items of meta-data can also be retrieved, such as the type of the underlying data.

Using a Recordset Object



```

_RecordsetPtr sRecordset;

STDMETHODIMP CPhones::CreateRecordset()
{
    _ConnectionPtr sConnection("ADODB.Connection");
    sConnection->Open(fileDSN, (BSTR)0, (BSTR)0, -1);

    sRecordset.CreateInstance("ADODB.Recordset");
    sRecordset->put_CursorLocation(adUseClient);
    sRecordset->Open("SELECT ID, Name, Number FROM Phones",
                    (IDispatch*)sConnection,
                    adOpenStatic,
                    adLockBatchOptimistic,
                    adCmdText);

    return S_OK;
}

```

364

The above code shows how to open a client side static cursor as a recordset object.

Firstly, we create a smart pointer **sConnection** that encapsulates a connection object. The smart pointer is then used to open the connection using a file DSN:

```

_RecordsetPtr sRecordset;
_ConnectionPtr sConnection("ADODB.Connection");
sConnection->Open(fileDSN, (BSTR)0, (BSTR)0, -1);

```

To create a recordset object we write:

```
sRecordset.CreateInstance("ADODB.Recordset");
```

We can now decide whether to use a server side or client side cursor. In this example we intend to use a client side cursor so we write:

```
sRecordset->put_CursorLocation(adUseClient);
```

Now we open the recordset using an SQL query:

```

sRecordset->Open("SELECT ID, Name, Number FROM Phones",
                (IDispatch*)sConnection,
                adOpenStatic,
                adLockBatchOptimistic,
                adCmdText);


```

Because we have opened a client side recordset we can disconnect the recordset from the database while we utilise the recordset:

```
sRecordset->putref_ActiveConnection(0);
```

Disconnecting the recordset is highly desirable in distributed systems because it minimises the time connected to the server database and allows us to develop scalable applications.

CursorType



<i>CursorType</i>	<i>see changes by others</i>			<i>CursorLocation</i>		<i>movement</i>	
Static				client	server	forward	back
Forward-only					server	forward	
Keyset	changes		deletes		server	forward	back
Dynamic	changes	adds	deletes		server	forward	back

365

Cursor types specify how a set of rows returned from a database query can be navigated. They also determine whether updates performed by other users of the database become visible in the recordset. In ADO Cursor type is a property of the Recordset object. Different cursors will provide different functionality.

A **Static** cursor is a static copy of the result of the query. Any changes made to the underlying data by other users of the database are not visible in the recordset. Note that it is the cursor that is static, the data may still be updated and changes written back to the database. A static cursor supports scrolling forwards and backwards through the recordset.

A **Forward-only** cursor is identical to a static cursor, except it does not support scrolling back through the recordset. Forward-only cursors are often used for generating reports, or populating fields on a form.

A **Dynamic** cursor allows updates to data to be visible to the recordset. It also allows the members of the recordset to change. This could happen if a query has been executed and another user adds records to the database which satisfy the query criteria.

A **Keyset** cursor is one where row membership is fixed when the query is executed. Any additions or deletions that would affect the query result are not propagated to the recordset. Data updates are visible. As the keyset only actually contains the keys, and a subset of records are retrieved at any one time, updates to the underlying data can be made visible to the recordset.

Providers are not required to support all cursor types and may substitute a different cursor type when the query is executed. In this case the CursorType property will be set to the correct value for the actual cursor type returned.

CursorLocation



- **Property of Recordset and Connection objects**
- **Specifies the location of the cursor engine**
- **Client side**
 - **adUseClient or adUseClientBatch**
 - **Uses a client-side cursor**
 - Supplied by local cursor library
- **Server side**
 - **adUseServer**
 - **Uses a server-side cursor**
 - Supplied by provider or driver
 - **Disconnected recordsets not supported**

366

The CursorLocation property specifies whether a client-side or server-side cursor engine will be used. This may affect the available cursor types for given provider. Both adUseClient and adUseClientBatch evaluate to the same constant and give a client-side cursor implementation. A client side cursor may be more efficient and/or provide more features than a server side cursor, especially in the area of static or forward-only cursors. Client side cursors are provided by a local cursor implementation.

Server side cursors are provided by either the OLE DB provider or the driver. Disconnected recordsets are not supported with server-side cursors. Some cursors, such as Dynamic cursors are easier to implement as server side cursors.

LockType



- **Property of Recordset**
 - Parameter to Recordset.Open
- **Specifies locking semantics for query or command**
- **Read only**
 - adLockReadOnly
 - Data cannot be updated
- **Pessimistic**
 - adLockPessimistic
 - Locks records on edit
- **Optimistic**
 - adLockOptimistic
 - Locks records only on Update
- **Batch optimistic**
 - adLockBatchOptimistic



367

The LockType property specifies the locking semantics for a query or command. For a straightforward query, where data does not need to be updated, adLockReadOnly should be used. This will allow other users to update any records returned by the query. If a recordset needs to be updated then several options are available. adLockPessimistic will lock the current record when any Field value is changed, the record will remain locked until Update or CancelUpdate are called. With adLockOptimistic, the record is only locked when Update is called, and is unlocked when Update completes.

adLockBatchOptimistic allows multiples records to be modified. The records are written back to the database when UpdateBatch is called. The records are only locked briefly during the UpdateBatch call.

adLockBatchOptimistic is required (along with several other settings) if the Microsoft Client Cursor Engine is needed. This cursor engine allows batch updates and disconnected recordsets.

Some experimentation may be needed with regard to locking semantics. adLockPessimistic is the safest option to use, but may have a detrimental affect on database performance. However, adLockOptimistic may cause problems if there are a significant number of clients performing updates to the database.

Marshalling Recordsets (1)



- **Recordset objects can be marshalled into client address space**
 - **By value**
 - **Uses the Microsoft Client Cursor Engine**
 - Batch updates
- **Specify a recordset as a return value**
 - **VB - Public Foo as ADOR.Recordset**
 - **IDL - void Foo([out, retval] _Recordset **ppRS);**
- **Specify CursorLocation as adUseClient or adUseClientBatch**
- **Set LockType to adLockBatchOptimistic**
- **Set CursorType to adOpenStatic**

368

One of the key features of ADO is the ability to pass Recordset objects, BY VALUE, across process boundaries. This allows the client to scroll through the data without any cross-process method calls, as all the data is in the client's address space. The data can also be updated and then sent back to the server process. This is known as batch updating.

There are several things that need to be done to ensure that a recordset gets marshalled correctly into the client address space. Firstly a method must be defined to return a `_Recordset` interface (or `ADOR.Recordset` type in VB). Then before the Recordset object is opened the `CursorLocation` property must be set to `adUseClient` (or `adUseClientBatch`). When the `Open` method is called, the `LockType` must be `adLockBatchOptimistic` and the `CursorType` must be `adOpenStatic`.

After the Recordset is opened it can then be passed back to the client and the results will be marshalled into the client address space. Note that this is not usually the case with COM objects, usually an interface pointer is marshalled across to the client, but the data remains in the server address space.

The disconnected recordset is populated asynchronously, so the `RecordCount` property may not be accurate when the recordset is first accessed.

Marshalling Recordsets (2)




- **Disconnected recordsets can be updated**
- **Marshaled back to server**
- **Specify MarshalOptions**
 - Property of Recordset object
- **To return all rows to server**
 - `adMarshalAll`
 - Default
- **To return only modified rows**
 - `adMarshalModifiedOnly`
- **Passed as object**

369

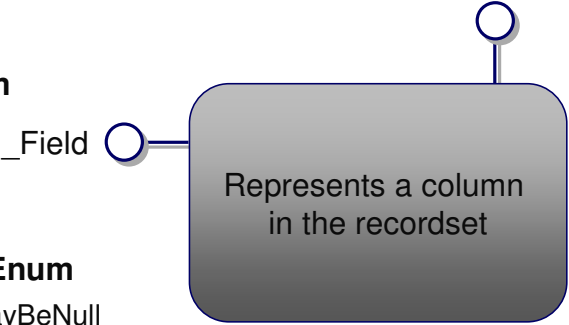
Having got a disconnected recordset in the client address space it can then be scrolled through and/or modified. The updates can then be sent back to the server by calling the `UpdateBatch` method. By default all records will be sent back to the server. Alternatively, the `MarshalOptions` property can be set to `adMarshalModifiedOnly`, which will ensure that only records which have changed will be sent back to the server.

An alternative to called the `UpdateBatch` method is to pass the recordset object back to the server by passing it as a method parameter. In MTS this may result in one method that returns the disconnected recordset and a second method that takes a recordset as a parameter and writes the changes back to the database.

Field Object



- **Value** P
 - Allow data to be read or written
 - Default property
 - Addressable by index or name
- **Name** P
 - Returns the name of the database field
- **Type** P
 - Returns a **DataTypeEnum**
- **Attributes** P
 - Characteristics of field
 - Returns a **FieldAttributeEnum**
 - adFldUpdatable, adFldMayBeNull



370

The Field object contains the actual values of the database field. Its value can be read or written through the Value property. The Value property is the default property of the Field object which means that values can be read/written by using an index or name on the Fields collection e.g.

```
rs.Fields(0)=10
x=rs.Fields("apt_id")
```

The Name property returns the name of the underlying database field. The Name property is read-only.

The Type property describes the underlying data type, while Attributes returns information about what can and can't be done with the field. As the Field object contains all this information, Recordsets are self-describing. A generic function could be written to query and display any arbitrary database.

Using Field Objects



```
STDMETHODIMP CPhones::GetRecords(BSTR *pRecords)
{
    _bstr_t result;
    sRecordset->MoveFirst();

    while(!sRecordset->ADOEOF)
    {
        FieldsPtr sFields = sRecordset->GetFields();
        _bstr_t ID_Field = sFields->GetItem("ID")->GetValue();
        _bstr_t Name_Field = sFields->GetItem("Name")->GetValue();
        _bstr_t Number_Field = sFields->GetItem("Number")->GetValue();
        result += ID_Field + Name_Field + Number_Field;
        sRecordset->MoveNext();
    }

    *pRecords = result.copy();
    return S_OK;
}
```

371

You see the real power of smart pointer when working with the Fields collection of a recordset. Without the use of smart pointers the code above would triple in size and become very difficult to read. By using smart pointers we end up with intuitive code.

Most of the above code is self evident, but we note how to obtain the fields collection object:

```
FieldsPtr sFields = sRecordset->GetFields();
```

This collection defines all the data for the current record. We can extract a given field, say the BSTR for the ID field with the code:

```
_bstr_t ID_Field = sFields->GetItem("ID")->GetValue();
```

Missing and Default Parameters



- **Missing Parameters**
 - Visual Basic allows default and missing parameters in methods.
 - In C/C++, all operands must be specified.
- **Default Variant Parameters**
 - vtMissing
- **Default BSTR Parameters**
 - `_bstr_t(L"")` or (BSTR) 0
- **Default IDispatch Parameters**
 - (IDispatch*) 0

372

Usage Suggestions



- **Use Connection objects in methods that update a single database**
 - Execute method
- **Use Command objects in methods that update multiple databases**
 - Execute method
- **Use Recordsets for query methods**
 - Disconnected if updating is needed

373

Finally, some recommendations for when each ADO object is appropriate.

Where a method updates a single database with a particular statement, Connection objects should be used. A Connection object refers to a single database. SQL INSERT, UPDATE and DELETE statements can be issued on the Connection.

Where an update is to be performed on two or more databases, a Command object may be more efficient, especially if parameterised queries are used. The Command object can be associated with different Connection objects and run against the underlying database.

Recordsets should be used for methods that perform Queries. If the client will be performing updates to the database, then use disconnected recordsets. Make the changes at the client then pass the recordset back to an MTS object via a method call. Do not use the UpdateBatch method, as if you do the changes to the database will not be transacted.

Summary



- **ADO is the preferred database-access method**
- **Simple object model**
- **Flexible and comprehensive**
- **Works well under MTS**

374