

Quick Start Guide to SQL Server 7

By Karl Moore

Introduction

There comes a time in every programmers life when your voice drops a pitch, hairs sprout in the most unexpected of places – and you realise the need to move onto a bigger and better database system.

Most of us start off dabbling with Access databases. They're fun, cheap and easy to maintain. But there comes a time – I call it *The Change* – when they're just not enough.

You need something more powerful, more expensive, more professional looking. And often, that something is SQL Server.

This four-part series provides quick, no-nonsense, step-by-step instructions for getting started with all editions of SQL Server 7.0. It doesn't delve into its history, it doesn't stop to evaluate whether or not this is actually the database system for you, it doesn't babble.

This guide is for those graduating from Access, needing to start using SQL Server – and *fast*.

It covers all the SQL Server essentials, alongside chunks of relevant Visual Basic and ADO code. From table creation right through to stored procedures, it's all here.

Remember, this guide is designed for printing. Either print each page direct from your browser, or get the friendlier, more compact version by clicking our print link to the right. Unfortunately this guide isn't currently available as one big download.

If you're looking for further help or assistance with problems, try checking out the Books Online application, under SQL Server's entry on the Start menu. If you can't find your answer there, post a question on our hyperactive bulletin boards at forums.vb-world.net.

Enough chit-chat, it's time to get serious. The topic is SQL Server, we're wanting just the facts, ma'am... and your time starts... **now!**

Guide Index

Databases:

- Launching your Server Taskpad
- Creating a Database
- Viewing your Database in Enterprise Manager
- Moving your Access database to SQL Server

Tables:

- Viewing your Tables
- Creating a Table
- Altering your Table
- Entering data into your Table
- Viewing your Table Properties

Data Types:

- List of all SQL Server data types

Security:

- Creating a Login
- Granting access to a user

Indexes:

- Creating an Index
- Running the Index wizard
- Managing your Indexes
- Running the Index Tuning Wizard

Relationships:

- Creating a relationship

VB Code:

- Skeletal ADO data access code

Views:

- Creating a View
- Editing your Views
- Granting user access to a View

Deleting, Renaming:

- Deleting an Object
- Renaming an Object

Defaults:

- Creating a Default
- Binding a Default
- Editing a Default

User Defined Types (UDTs):

- Creating a UDT
- Implementing a UDT

Rules:

- Creating a Rule
- Binding a Rule
- Editing a Rule

Transactions:

- How they Work, with VB Code

Triggers:

- Managing Triggers
- Sample Trigger Code

Stored Procedures:

- Creating a Stored Procedure
- Editing a Stored Procedure

Stored Procedure Examples, with VB code:

- SELECT Stored Procedure
- UPDATE Stored Procedure
- DELETE Stored Procedure
- INSERT Stored Procedure
- Output Parameters

Databases

Databases are a holding place for information.

Within SQL Server, a database may hold numerous tables, views or stored procedures, among other items.

We'll cover all of these later, just remember that SQL Server has features that Access doesn't have (such as stored procedures), and at the same time, Access sports certain qualities not found in SQL Server (such as Forms).

Another core difference is that Access is essentially just an .MDB file, whereas SQL Server is 'live' and running on the server.

In this section, we'll learn how to create an empty database – the holding point for tables and so on – plus port any existing Access databases across to SQL Server via something called the Upsizing Wizard.

Launch your Server Taskpad:

- Start Enterprise Manager, the front-end for SQL Server
 - *Programs/Microsoft SQL Server 7.0/Enterprise Manager*
- Expand the SQL Server Group item to view all connected systems running SQL Server
- Dip into the computer you want by double-clicking on its icon

To create a Database:

- Launch your Server Taskpad (as before)
- Click the 'Set Up your Database Solution' icon
- Click the 'Create a Database' icon to run the Wizard
- On the first introductory screen, click Next
- On the next screen, give your database a name and location, then click Next
 - *It's best to give your database a name without spaces, to avoid confusion and errors later*
 - *The database file location will be the place your actual tables, data within them, and so on, are stored*
 - *Every action in your database is recorded in a transaction log. So, if you have a hardware failure or major database corruption, you can restore data from that transaction log. For greatest peace of mind, change the transaction location to an external drive – one different from the location of the main database files*
- This screen simply asks you to set the initial starting size of your database. Change if required, then click Next
 - *With Access, database sizes grow as more information is put into them. With SQL Server, you set a certain amount of disk space to one side, ready to hold future data*
 - *When you need to store more information, previously you had to manually 'expand' the database. However SQL Server 7 can now automatically do that expanding for you, as we'll see on the next screen*
- On this screen, select your database growth options. Click Next when finished.
 - *The default options dictate that when the current database is 'full', it allocates another block of hard disk space for the database to use.*
 - *It's recommended you stick with the defaults, unless you have specific requirements*
- The next two screens ask you for the initial size and growth rates of your transaction logs. Click Next when finished
 - *Once again, it's recommended you accept the defaults unless you have specific requirements*
- On the final Wizard screen, review the description of your new database, move back and forth to implement any changes, then click Finish when complete
- After the database is complete, you may be prompted to create a maintenance plan for the database
 - *The maintenance wizard allows you to schedule database backups and so on. You may follow this through if you wish*
 - *We will not be covering maintenance in this quick start guide*

To view your database in Enterprise Manager:

- Launch your Server Taskpad (as above)
- Double-click on your server name, to the left of your screen
- Double-click on the 'Databases' folder
 - *You should see a list of databases on the screen, including your own. Many of these are samples that ship with SQL Server.*
- Double-click on your database name
 - *If you're new to this, don't worry about some of the mysterious items listed under your database such as Rules and Stored Procedures. We'll cover the most important of these later*

It's also worth noting that Microsoft Access 2000 ships with an Upsizing Wizard that can automatically port your .MDB database over to SQL Server.

To move your Access database to SQL Server:

- Launch Microsoft Access 2000
- Click 'Tools', 'Database Utilities', 'Upsizing Wizard'
 - *If you're using Access 97, you probably won't have the Upsizing Wizard we're looking for. To download it, visit www.microsoft.com/accessdev/prodinfo/AUT97dat.htm*
 - *For all other versions, it's advisable to upgrade your database to Access 97/2000 and then use the Upsizing Wizard from there*
 - *Unlike Access, SQL Server doesn't use 'forms'. Only your tables, queries and relationships will be ported across*
- Follow the on-screen prompts

Tables

The most important objects within your database are tables.

These are a lot like Excel spreadsheets, each storing rows of information, such as a customer order or shipping address.

Typically, you would utilise normalisation rules when creating the tables, meaning you should ensure each table has no repeating columns, utilises primary and foreign keys where appropriate, and so on.

To view the Tables:

- View your Database in Enterprise Manager (as before)
- Click the 'Tables' image
 - *Any tables displayed beginning with 'sys', as well as 'dtproperties' are SQL Server's own. It uses these tables to store information about your database*

To create a Table:

- View the Tables (as before)

- Click 'Action' in the toolbar, select 'New Table'
- Enter a Table Name when prompted
 - *You may wish to use a table naming convention here. Many opt for tblTableName, whilst others prefer TableName_t. It's your call*
- Define your fields, row by row
 - **Column Name** – Enter the name of this field. You may wish to use naming conventions here, depending on the data type
 - **Datatype** – Select the required data type (full explanatory list on the next page)
 - **Length** – This isn't always editable. It specifies the length of your data type (eg, char data type, 15 characters in length, or varchar data type, maximum 15 characters in length)
 - **Precision** – Applies to numeric data only. This isn't always editable. Specifies the maximum number of digits in the number (eg, the number 539.154 has a scale of six). Almost a length value for numeric data types
 - **Scale** – Applies to numeric data only. This isn't always changeable. Specifies the maximum number of digits to store to the right of the decimal point (eg, 12.528 has a scale of three)
 - **Allow Nulls** – Check if Null values (ie, nothing) are allowable in this field. Otherwise, uncheck
 - **Default Value** – A default value to be placed into this field when a new row is inserted
 - **Identity** – If this will be an identity column (ie, you want SQL Server to place a unique number into the field), check this box
 - **Identity Seed** – If this is an identity column, enter the starting point for the unique numbers. By default, this is set at one
 - **Identity Increment** – If this is an identity column, with each new row, the unique ID number will be incremented. Typically, it is upped by one each time. If you wish to change that figure (ie, increase numbers by ten each time), enter the incrementing value here
 - **IsRowGuid** – Check if you want to automatically insert a Globally Unique Identifier (GUID) into this field. These are guaranteed unique IDs and no two in the world should ever be the same. An example of a GUID is {B31C08E5-2A56-11D4-B09E-CFB9E165D54C}. To choose this option, you must have a Data Type of Unique Identifier
- When finished, press CTRL-F4 to close the table
- If prompted to save, click Yes/No

To alter your Table design:

- View the Tables (as before)
- Right click on your table
- Select 'Design Table'
- Make changes
- Press CTRL-F4 to exit
- If prompted to save, click Yes/No

To enter data into your Table:

- View your Tables (as before)
- Right click on your table
- Select 'Open Table', 'Return All Rows'
- Start entering data, row by row
 - *Note that this table viewer is a little buggy. For example, if your table specifies that a GUID should be inserted into a field, it will appear as '<Null>' or as a bunch of zeros until you hit the Run button (exclamation mark icon on the toolbar)*

To view Table Properties:

- View the Tables (as before)
- Single click on your table
- Press ALT-Enter

Data Types

Each field in a table needs to be of a specific data type. The following list describes each of them, starting with the most common.

Data Type	Description
Varchar	Akin to the VB String data type. A variable length string containing up to 8,000 characters. Non-Unicode (doesn't support International characters)
Char	Akin to the VB String data type. A fixed-length string containing up to 8,000 characters. Non-Unicode.
Int	Akin to the VB Long data type. A whole number ranging from -2,147,483,647 to 2,147,483,647.
Smalldatetime	Akin to the VB Date data type. Holds any date and time combination between January 1 st , 1900 to June 6 th , 2079, with an accuracy of one minute.
Smallmoney	Akin to the VB Currency data type. Holds a currency amount between -214,748.3648 to 214,748.3647
Smallint	Akin to the VB Integer data type. Holds a whole number between -32,768 to 32,767.
Tinyint	Akin to the VB Byte data type. Holds a whole number from between 0 to 255.
UniqueIdentifier	A 128-bit, Globally Unique Identifier (GUID)
Text	A variable-length string holding up to 2,147,483,647 characters.
Money	Akin to the VB Currency data type. Holds a currency amount ranging from -922,337,203,685,447.5808 to 922,337,203,685,447.5807

Datetime	Akin to the VB Date data type. Can hold any date and time combination between Jan 1 st , 1753 to December 31 st , 1999, with an accuracy of 3.33 milliseconds.
----------	--

[Check us out next week for the second part in our SQL Server print-and-collect series. We'll be covering; security, indexes, relationships and Visual Basic data access code]

Granting Permissions

Once you have information in a database, you'll probably want to access it from your application.

But to gain access, you'll first require a username and password login. Then you'll need to grant that login special rights to your tables and so on.

We'll be covering all that in this section.

To create a Login:

- View your Database in Enterprise Manager (as before)
- Click 'Set Up your Database Solution'
- Select 'Create a Login'
- Hit Next at the Welcome screen
- Select the SQL Server authentication mode, then click Next
 - *If you're running the Desktop edition of SQL Server on Windows 95/98, you will only be able to select SQL Server authentication. This means SQL Server handles all the usernames, passwords and other security issues*
 - *If you're running Windows NT, you will also have the option of using Windows NT authentication. This means you can grant certain NT users direct access to databases, based on their Windows logon, without requiring a separate username and password for SQL Server*
 - *These instructions give advice on creating a login for SQL Server authentication mode*
- Enter a login name, then type in the password twice. Click Next when finished
- If you're creating an account for a special user, such as a developer or such like, select the appropriate security roles here. Otherwise, simply ignore. Click Next when finished
- Check the databases you wish to grant this new user access to, then click Next
 - *Allowing a user access to a database means just that – they get access to the database, **not** the individual objects within it, such as the tables. You need to do this separately (as below)*
- Hit Finish to create the login

To grant Table access to a user:

- View your Table Properties (as before)
- Click the 'Permissions' button
- Tick the boxes dependant on which permissions you want to grant the user
 - *To allow the user to view information in the table, check 'Select'*
 - *To allow new records to be added, check 'Insert'*
 - *To allow the deletion of records, check 'Delete'*
 - *To allow the update of records, check 'Update'*
 - *When granting permissions for stored procedures, which we'll come across later, you'd typically check the 'Exec' (execution) box*
- Click OK on the Permissions form, then OK on Table Properties

You will typically assign a username and password combination for either an entire application (ie, one generic login with appropriate permissions), or on an individual user basis.

Indexes

Most tables in your database will typically hold at least one index.

A table index is rather like the index of a book. It allows for the fast retrieval of specific information.

Adding an index to a field makes SQL Server go and organise all the information within that field, sorting and making it quicker to access that rows' data in future.

But unfortunately indexes in SQL Server aren't quite as simple as Access. Here, we have two different types of index:

- **Clustered** – This type of index sorts the entire table based on one field. Imagine it as a phone book; it organises the Surname field in ascending order and can therefore find rows quicker based on that field. Of course, if it were just to search through a random list of data, it would be much slower. Note though, you can only have one clustered index per table
- **Non-Clustered** – This is a strange index. Let's say you add this index to the Surname field. This index stores all of the Surnames in ascending order somewhere else, along with a pointer to the original row. So when you search for a particular Surname, it looks it up in the separate list of surnames, finds the pointer to the actual row, then speedily retrieves it. You can have more than one non-clustered index per table

You can use both clustered and non-clustered indexes within the same table. And even though SQL Server does a lot of behind-the-scenes work to implement these, you should note that this doesn't affect the way you retrieve records via Visual Basic.

It's also worth pointing out that you can involve more than one field within an index if you wish. For example, you may want to index both the Surname and Forename fields together – meaning searches on both of those fields will be lightning fast.

But adding too many indexes also slows down data addition, as after every new record insert, SQL Server needs to reorganise the index. So use them in moderation.

It's also worth noting that, just as in Access, indexes can be unique - or not. If an index is unique, it does not allow duplicates (sometimes called duplicate keys).

To create an Index:

- Open your table to alter the design (as before)
- Click the 'Table and Index Properties' button on the toolbar
- Select the 'Indexes/Keys' tab
- Click 'New'
- Select the Columns (fields) you want to add to your Index
 - *You can select more than one if you wish*
- Enter an Index name
 - *Alternatively, stick with the default*
- Check 'Unique' if you want this column to contain only unique values
 - *It will automatically select the 'Index' option button. This creates a unique index, whilst the 'Constraint' option simply stops duplicates. Stick with the default.*
- If you would like this to be a clustered index, check the 'Create as CLUSTERED' box
- To add another index, click 'New'
- Click Close when finished

To run the Index Wizard:

- Launch your Server Taskpad (as before)
- Click 'Tools' on the toolbar, then 'Wizards'
- Expand the 'Database' entry
- Select 'Create Index Wizard' and click OK
- Follow the on-screen prompts

To manage your Indexes:

- View your Tables (as before)
- Select the one you want to manage
- Right-click, select 'All Tasks', 'Manage Indexes'
- Use the 'New', 'Edit' and 'Delete' commands to alter as required
- Click Close when finished

If you're working on a large project and are unsure where you should add indexes, the Index Tuning Wizard can take frequently used SQL statements and decide for you. We won't be covering this feature in-depth here - for more information, check out Books Online.

To run the Index Tuning Wizard:

- Launch your Server Taskpad (as before)
- Click 'Tools' on the toolbar, then 'Wizards'
- Expand the 'Management' entry
- Select 'Index Tuning Wizard' and click OK
- Follow the on-screen prompts

Relationships

Relationships in a database allow you to define how information in one table links to information in others.

This is very easy using Microsoft Access, and not that much more difficult with SQL Server, despite a few changes in terminology.

Let's quickly define the two main types of relationship:

- **One-to-One** – In this type of relationship, you say that an entry in one table can only have one entry in another table. For example, an entry in the Employees table may only be allowed one entry in the CompanyCars table
- **One-to-Many** – In this type of relationship, you say that an entry in one table can have many different entries in another. For example, an entry in the GeneralOrder table may have many related entries in the IndividualOrderItems table.

You typically make these relationships work by giving each record a unique value of some sort. For example, say you run a veterinary hospital. You'd probably have a Customer table, containing your customer name and addresses, along with a unique CustomerID number.

You might also have a Pets table, containing the name and breeds of each individual pet the hospital deals with. You'd probably also throw a CustomerID field into that Pets table, so you can tell which customer owns which pet.

Adding relationships to your table ensures data integrity. If we didn't add any relationships to the above scenario, something might go wrong and a particular Pet record might be given an invalid CustomerID. Relationships protect that from happening. They could also stop someone accidentally deleting a Customer that has related Pet records.

In this case, you might consider adding a one-to-many relationship between the main Customer table and the Pets table.

In other words, the relationship is between the CustomerID field in the Customer table (the 'one' part of the relationship) and the CustomerID field in the Pets table (the 'many' part of the relationship).

Then if you tried to add a pet with a CustomerID that didn't exist in the Customer table, the record would be rejected. And these same rules could also stop customers from being deleted when they still have entries in the Pets

table.

In brief once more; *relationships help maintain data integrity.*

To create a relationship:

- Ensure both the tables you wish to join have the necessary keys and are of the same base data type
 - A 'primary key' ensures there are no duplicates in a particular field
 - If you're creating a one-to-many relationship, the main table field(s) needs to be a primary key (eg, CustomerID)
 - If you're creating a one-to-one relationship, the field(s) from both tables involved need to be primary keys
 - To add a primary key, edit your table design, highlight the field(s) that you wish to change to primary key status, then click the key image on the toolbar
- View your database in Enterprise Manager (as before)
- Right-click on 'Diagrams' and select 'New Database Diagram...'
 - A Database Diagram in SQL Server is like the Relationships screen in Access
- Cancel the Wizard
- Click the 'Add Table' button on the toolbar
- Add the desired tables, then hit Close
 - You may find the tables have hidden themselves under each other. To view properly, simply drag them apart
- Highlight the field(s) you want to be involved in the relationship from the main table
 - By clicking the small box to the left of the field
- Drag and drop the adjacent box on the field you want to create the relationship with, in the secondary table
- In the box that appears, check the relationship parameters, then click OK
 - You will typically have one field from the primary key table, another from the foreign key table
 - It is recommended you accept the default relationship options
- Hit ALT-F4 to close the Database Diagram
- Give the diagram a name, then click OK
- If you are given warnings stating existing data violates the relationships you attempted to put in place, sort it out
- If prompted that the specified tables will be saved to your database, click Yes to continue

Accessing from VB

Naturally, you can use the DAO and ADO controls to access your SQL Server data. But most users prefer to get at it via pure code – after all, not only does it reduce the need to use a form and bound controls, it's also a lot more flexible.

We're now going to look at a skeletal piece of commented ADO code for accessing your SQL Server records from within Visual Basic. Note that you must have a reference (Project, References) to the Microsoft ActiveX Data Objects

Library before running this code.

```
Private Sub Command1_Click()  
  
Dim objConn As New ADODB.Connection  
Dim objRS As New ADODB.Recordset  
Dim objErr As ADODB.Error  
  
objConn.Open _  
"Driver=SQL Server;Server=COLOSSI;Database=Blossom;User ID=KarlMoore"  
'Open the SQL Server connection  
  
objConn.CursorLocation = adUseClient  
  
If objConn.State = adStateOpen Then  
    'If everything is OK and we have a connection  
  
        objRS.CacheSize = 10  
  
        objRS.Open "Select * from tblCustomers", objConn, adOpenStatic  
  
        ' Open the recordset object, selecting everything  
        ' from tblCustomers - passing our objConn connection  
  
        If Not objRS.EOF And Not objRS.BOF Then  
            ' If not at the End Of File (EOF) nor  
            ' Beginning Of File (BOF) - in other words,  
            ' if we actually have records here - then...  
  
                MsgBox objRS.Fields("CustomerID")  
                ' Display one of our fields just to prove  
                ' everything went A-OK  
  
                objRS.Fields("CustomerID") = 123  
  
                objRS.Update  
  
            End If  
  
            objRS.Close  
            ' Close the recordset  
  
        Else  
            'If the connection didn't open A-OK, then  
  
                For Each objErr In objConn.Errors  
                    MsgBox objErr.Description  
                Next  
  
                ' Display all the errors in the connection object  
  
            End If  
        End If  
    End If  
End Sub
```

```
Set objRS = Nothing
objConn.Close
Set objConn = Nothing

'Tidy up the references, etc.

End Sub
```

Most of this is pretty standard stuff. However there are a few lines of code you *may* wish to fine tune to suit your individual needs, particularly if you're working in a multi-user, real-time environment.

Unfortunately, this is neither an ADO nor general data access tutorial - and some of the following issues are simply raised for completeness. If it's your first encounter, you may find it all a little strange. Thankfully the ADO help file can help out, with its own mini tutorial. To view it, find and run the file 'ADO210.CHM' or 'ADO200.CHM'. Alternatively, just stick with the template.

CursorLocation Property

```
objConn.CursorLocation = adUseClient
```

First off, we have the `CursorLocation` property. The cursor represents where in the recordset you currently are. It's a little like Word's own flashing cursor, which shows you where you are in the current document.

You can set this to either:

- **adUseServer** – This keeps the cursor with the server, a real resource hog if you have plenty of users. However, when used with certain types of cursor (below), it does have the advantage of being able to - spot newly added records, remove fresh deletions from your recordset and so on. But you use it at a price.
- **adUseClient** – This keeps the cursor on the client side, the user's computer. This removes workload from the server but doesn't allow you to view new records that match your criteria, etc. Still, most of the time, such features aren't required

If you're in doubt, go for `adUseClient`.

Cursor Type

```
objRS.Open "Select * from tblCustomers", objConn,
adOpenDynamic, adLockOptimistic
```

The third argument in this code asks for the type of cursor you want. You can replace the current `adOpenDynamic` with any of the following:

- **adOpenForwardOnly** – This cursor allows you to simply scroll forward through the recordset, but there's no going back. This option is very resource friendly

- **adOpenKeyset** – This cursor allows you to move both backwards and forwards through the recordset. It maintains a live link with the database to ensure all fields are kept up-to-date (ie, they reflect the changes of other users), however it doesn't detect any externally added records. The keyset cursor maintains a live connection to the database and is quite a resource hog
- **adOpenStatic** – This option grabs a copy of the records. You can't view any changes, new records or deletions by other users, though you *can* make edits yourself. This cursor is pretty resource friendly
- **adOpenDynamic** – Using a dynamic cursor, you can view all changes, additions and deletions by other users. It maintains a live conversation and so is pretty resource expensive

Lock Type

```
objRS.Open "Select * from tblCustomers", objConn,  
adOpenDynamic, adLockOptimistic
```

That fourth argument in this code asks for a 'lock type'. The current `adLockPessimistic` is interchangeable with any of the following:

- **adLockReadOnly** – Read only. The data retrieved cannot be modified. Very resource friendly, but not always practical
- **adLockPessimistic** – A lock is placed on records as soon as you start editing them and released when the Update or Cancel methods are called
- **adLockOptimistic** – This option means that when you run the Update method, the record is locked, updated and then released, all in one swoop. If someone else changed the record before you started editing it, an error will occur
- **adLockBatchOptimistic** – Whereas `adLockOptimistic` allows you to edit a record 'offline', then update it in one swoop, `adLockBatchOptimistic` allows you to do this with more than one record. You can edit the records in the recordset, then do a complete database update on all those changed by running the UpdateBatch method. Any errors will be returned to the Errors collections of the connection

Cache Size

```
objRS.CacheSize = 10
```

Another interesting recordset property is `CacheSize`. This determines how many rows of data are buffered on the client machine.

Increasing this from the default of one allows the server to work more efficiently by returning a whole bunch of rows to the client at once, as opposed to one per shot.

However, don't forget - this means the information you store locally may not always be the most up-to-date from the server. So in a multi-user environment, this could lead to update conflicts.

Creating Views

Views are essentially the same as Microsoft Access queries.

You tell the database what information you want to retrieve, then give that query ('view') a name.

The data from this view can then be retrieved via a piece of code such as that in the 'Accessing from VB' section. Instead of doing something like this in Visual Basic code:

```
objRS.Open "Select * from tblCustomers", objConn
```

You simply alter the SQL Statement to the name of the view, like this:

```
objRS.Open "MyView", objConn
```

This means you're not simply storing all your SQL statements within your application, but rather in one central depository, your database.

Also, you can grant permission for certain users to access your view, but not the underlying table. That enables you to stop unauthorised users gaining access to privileged information, such as the salary field in an employee table.

To create a View:

- View your Database in Enterprise Manager (as before)
- Click 'Views'
 - *The existing views are used by SQL Server to maintain your database. Do not remove them.*
- Right-click 'Views', then select 'New View'
 - *Depending on which version of SQL Server you are using, the screen you see may vary.*
- Type your SQL statement in the box currently containing the skeletal "SELECT FROM"
 - *Alternatively, you may use the Access-like tools on the same screen to build your view*
 - *To test your SQL statement, hit the exclamation button on the toolbar*
- Hit ALT-F4 to close the design view window
- Respond to any prompts to change, saving with an appropriate name

To edit a View:

- View your Database in Enterprise Manager (as before)
- Click 'Views'
- To re-enter the design view, right-click and select 'Design View'
 - *Alternatively, double-click to edit the raw SQL statement, without all the visual frills*
- Press ALT-F4 when finished, responding to any prompts

To grant View access to a user:

- Follow all the steps for granting Table access to a user (as before), but using your view as opposed to a table

Deleting, Renaming

Sometimes you need to delete or perhaps rename an object, such as a view or table. This is a very simple process.

To delete an Object:

- Locate the Object within Enterprise Manager
- Right-click on the Object
- Select Delete
- Check the items to be deleted in the prompt, then hit 'Drop All' when ready

To rename an Object:

- Locate the Object within Enterprise Manager
- Single-click on the Object
- Press F2
- Type in a new name
- Press Enter
- If prompted that the rename may cause problems, decide what to do

Defaults

Defaults are one of the simplest concepts within SQL Server. They're just default values set to one side, which you can tie to particular fields.

You can add defaults whilst designing your tables. However the default method demonstrated here allows you to store the default value for multiple fields in one single location, allowing for easier maintenance.

To create a Default:

- View your Database in Enterprise Manager (as before)
- Right-click the grey 'Defaults' icon, and select 'New Default'
- Type in a Name for your Default
- Specify a Default Value
 - *If you're specifying a default text value, enclose it in quotation marks*
 - *Dates and numbers do not require surrounding quotation marks*
- Click OK to save

To bind a Default to a Field:

- Create your Default (as before)
 - *You can't bind a default until you've first saved and closed it. So*

ensure you save and close it, then open again at the next step

- Open your Default by double-clicking on it in Enterprise Manager
- To bind a Default to a field, click 'Bind Columns', select your table, add the individual fields, then click OK
- To bind a Default to a particular user defined type (we'll explain these on the next page), click 'Bind UDTs', then check the 'Bind' box for all those to be affected
 - *Checking the 'Future Only' box means existing columns using the UDT won't inherit the new default*
- Click OK to save

To edit a Default:

- Open your Default, remove all dependencies of the Default, then close it by clicking OK
 - *In this instance, dependencies are items that relies on this default to work, such as a bound column*
 - *To view dependencies, right-click on your default, select 'All Tasks', 'Display Dependencies'*
 - *To remove dependencies, open your Default, then click on both the 'Bind Columns' and 'Bind UDTs' buttons, removing all items that rely on the default*
- Open your Default again and edit as required
- Click OK
- Open your Default yet again and rebind the columns as before, closing and saving when finished

User Defined Data Types

User Defined Data Types are simply customised versions of the basic data types.

As an example, you might create a particular User Defined Data Type (UDT) to hold a postal code. In England, that would probably be defined as a char (string) of seven characters long that doesn't allow nulls.

When designing your tables, you'll find any UDTs you created among all the regulars.

The main purpose of UDTs is to allow for easier, centralised maintenance of data integrity.

After all, if the postal code system – or something else slightly more volatile – changes, you only have to update your one UDT, not a myriad of random tables.

Another magical advantage is that you can bind defaults (as per the last section) and rules (as per the next section) to specific UDTs.

To create a UDT:

- View your Database in Enterprise Manager (as before)
- Right-click the 'User Defined Data Types' icon, and select 'New User Defined Data Type'
- Type in a Name for your UDT
- Select a data type, length (if applicable) and check for whether or not you wish to allow null values
 - *None of this information can be changed after you have created the UDT*
 - *Also, it would appear that whether or not you check to allow null values or not, this option is ignored – meaning you must implement this functionality at table level.*
- Optionally, specify a Rule and Default
 - *The Rule and Default options can be altered after the UDT has been created*
- Click OK to save

To implement a UDT:

- Start creating your table (as before)
- On the field you want to use the UDT, select it from the list of available data types
- Continue the table creation process as normal

Rules

Rules are wonderful little widgets that allow you to dictate what sort of information can go into a particular field. They're what Microsoft Access calls a Validation Rule.

However in Access, you put the rules behind each individual field. In SQL Server, you create a rule, then tell the system which columns that rule applies to.

Here's an example of a rule: @value = 'P'

This tells SQL Server the field @value must equal 'P'. If the rule is evaluated as True (eg, the field @value is equal to 'P'), the field value is accepted. If the rule is evaluated as False (eg, the field @value is not 'P'), the field value is rejected.

Here's another example: @value >= £7000 AND @value < £25000

This allows any money amount greater than or equal to £7,000 and less than £25,000.

And one more example: @value IN ('1389', '0736', '0877')

This allows a field value of 1389, 0736 or 0877 – anything else gets thrown out the window.

Note that existing table data isn't checked by added rules, only updates and

new additions.

To create a Rule:

- View your Database in Enterprise Manager (as before)
- Right-click the yellow 'Rules' icon, and select 'New Rule'
- Type in a Name for your Rule
- Specify the Rule text
- Click OK to save

To bind a Rule:

- Create your Rule (as before)
- Open your Rule by double-clicking on it in Enterprise Manager
- To bind a rule to a field, click 'Bind Columns', select your table, add the individual fields, then click OK
- To bind a rule to a particular user defined type (as per the last section), click 'Bind UDTs', then check the 'Bind' box for all those to be affected
 - *Checking the 'Future Only' box means existing columns using the UDT won't inherit the new rule*
- Click OK to save

To edit a Rule:

- Open your Rule, remove all dependencies of the Rule, then close it by clicking OK
 - *In this instance, dependencies are items that rely on this rule to work, such as a bound column*
 - *To view dependencies, right-click on your rule, select 'All Tasks', 'Display Dependencies'*
 - *To remove dependencies, open your Rule, then click on both the 'Bind Columns' and 'Bind UDTs' buttons, removing all columns that rely on the rule*
- Open your Rule again and edit as required
- Click OK
- Open your Rule yet again and rebind the columns as before, closing and saving when finished

Transactions

Transactions allow you to complete numerous database operations as one whole unit.

Let's take the common example of bank transactions. Mr Bloggs walks into his local Natwest and hands over a cheque given to him by Mr Smith.

After this is tapped into the computer, your application may remove the money from Mr Smith's account... and then crash. That means Mr Smith has lost his money and Mr Bloggs is left wondering what's happened to his cheque payment.

Or perhaps your system first adds the money to Mr Smith's account, then attempts to deduct it from the account of Mr Bloggs. What happens if Mr Bloggs doesn't have adequate cash reserves? That means Mr Smith now has money in his account that shouldn't really be there.

All these problems are solvable with transactions. You can 'start a transaction', then take the money from Mr Smith, give it to Mr Bloggs, then 'commit the transaction'.

Committing the transaction tells SQL Server everything is *fine*.

However you can also 'rollback the transaction'. After deducting money from Mr Smith and adding it to the bank account of Mr Bloggs, you might experience an error. Maybe Mr Bloggs has since closed down his account, or something similar.

You can then 'rollback' (cancel) the transaction, meaning Mr Smith doesn't have any money deducted, nor Mr Bloggs have any money added.

In other words, both items of work are *either completed or rejected as one entire unit* – not individual database operations.

Let's now look at a simple Visual Basic transaction. This code attempts to add two extra records to the Jobs table. It then prompts the user if they want to commit or rollback.

If they commit, both records should be added. If they abort, neither should be added.

In real life however, these records would probably be in different tables. And transactions aren't limited to record adding. You can commit and rollback anything from stored procedures to table creation.

And don't forget that you wouldn't usually prompt a user to commit or rollback. You would typically commit or rollback in code depending on whether you received errors (eg, as a result of data integrity, etc).

Visual Basic Code

```
Private Sub Command1_Click()  
  
Dim objConn As New ADODB.Connection  
Dim objRS As New ADODB.Recordset  
  
'Objects to be used in this operation -  
'objConn, the basic connection  
'objRS, to hold our recordset  
  
With objConn  
    .ConnectionString = "Driver=SQL Server;Server=COLOSSI;" & _  
        "Database=PUBS;User ID=KarlMoore;Password=TEST"
```

```

        .Open
End With

'Get a connection to the database

objConn.BeginTrans

'Begin Transactions here!

objRS.Open "Select * from Jobs", objConn

'Open the recordset, selecting all from the Jobs table

objRS.AddNew
objRS.Fields("job_desc") = "My New Test Job"
objRS.Fields("min_lvl") = 10
objRS.Fields("max_lvl") = 20
objRS.Update

'Add a new record... and then...

objRS.AddNew
objRS.Fields("job_desc") = "My Second New Test Job"
objRS.Fields("min_lvl") = 20
objRS.Fields("max_lvl") = 30
objRS.Update

'... another!

If MsgBox("Do you want to commit?", vbYesNo + vbQuestion) = vbYes Th
    objConn.CommitTrans
Else
    objConn.RollbackTrans
End If

'Run the CommitTrans or RollbackTrans methods,
'depending on user response

objRS.Close
Set objRS = Nothing
objConn.Close
Set objConn = Nothing

'Close all references

End Sub

```

Triggers

Triggers give the developer more control over their data.

Not comparable with anything in Microsoft Access, triggers allow the developer to really control what information is allowed into the database, as well as what

happens with it.

Triggers help maintain data integrity.

They fire off after certain events, such as the addition, update or deletion of a record.

Let's look at a few possible trigger uses:

- **Enforce business rules**– if you have a few complex business rules that you can't handle via regular relationships and table constraints, you can throw them into triggers. For example, you might want to check every order to ensure the customer hasn't ordered more than three special offer video recorders
- **Log transactions** – Perhaps you want to keep a log of what is happening within the database. Every time a record is altered in any way, you can write code to throw a log entry into a separate table (perhaps even a separate database)
- **Maintain data integrity** – Relationships are really only triggers under another name. Every time you add or change records, a hidden trigger fires off and checks for data integrity. You can add your own using triggers, or perhaps implement Access features not supported by SQL Server relationships, such as Cascade Update and Cascade Delete

We'll attempt to cover the basics of implementing trigger statements in this section, though it's a complex topic and can be difficult to grasp. Don't worry if you don't understand it all at first.

To manage Triggers:

- View your Tables (as before)
- Right-click your Table
- Select 'All Tasks', 'Manage Triggers'

Let's now take a peek at a few sample triggers.

This trigger checks if a certain field in Table1 contains a particular value. If it does, that record is rejected:

```
CREATE TRIGGER MyTriggerName ON [TABLE1]

-- This says create a trigger called MyTriggerName to monitor Table1

FOR INSERT, UPDATE

-- And launch this trigger every time an insert or update --
-- is made on this table. The three default options are --
-- INSERT, UPDATE, DELETE - add or remove as appropriate --

AS
```

```
Declare @MyVariable as Char(10)

-- This declares a variable called MyVariable as Char(10), --
-- a string data type to hold a default ten characters --

Select @ MyVariable = Inserted.MyFieldName from Inserted

-- When inserting/update information, SQL Server creates a temporary
-- table (Inserted) to hold the information. When deleting, it uses
-- a table called Deleted. This bit of code uses an SQL statement --
-- to retrieve information from the Inserted table and place --
-- it into the MyVariable variable --

If @MyVariable = "Karl"

-- If MyVariable equals Karl then --

    Begin

        Raiserror('Invalid entry. Choose another name', 16, 1)
        -- Raise an error --

        Rollback Transaction
        -- Discard the record --

    End

-- Note that the Begin and End statements are just the --
-- boundaries of what should happen after the If statement --
```

Here's another example trigger:

```
CREATE TRIGGER SalaryLog ON [MyEmployees]
FOR UPDATE

-- Launch on an update --

AS

DECLARE @EmpName as VarChar(100), @EmpSalary as Money

-- Declare two different variables of different data types --

SELECT @EmpName = Inserted.EmployeeName,
@EmpSalary = Inserted.Salary
FROM Inserted

-- Grab the stuff being updated and place it into the variables --

INSERT INTO MyLog (Username, TheDate, Alteration)

Values (USER_NAME() , GETDATE(),
```

```
@EmpName + 'is now on ' + CONVERT(VarChar(10), @EmpSalary))

-- Insert the logged on database user and the date into the log tabl
-- Also, insert a description - @EmpName added to 'is now on ' addec
-- to @EmpSalary, which has been converted from the Money data type
-- to a varchar using the CONVERT function --
```

It's worth noting that triggers only fire off once even if you're altering multiple records. For example, you may be performing a mass update or delete – and if just one of those records violates the rules of your trigger and you issue a ROLLBACK TRANSACTION, it'll stop the whole lot.

Also, triggers can't be called manually from within Visual Basic. They're simply 'triggered' by SQL Server.

For more information on triggers, look up the topic in the Books Online reference that ships with SQL Server 7.

Stored Procedures

Stored Procedures (SPs) are chunks of code that reside on the server. Each SP can perform a wide variety of tasks.

You can call a stored procedure from within Visual Basic, perhaps passing a required argument or accepting a 'return value'.

Let's look at a few of the advantages using stored procedures gives us:

- **Easy Maintainability** – Stored procedures can get very complex, allowing you to moving difficult code from the client program and place it on the server. This allows for easier maintenance
- **Network Traffic** – Asking the server to do a chunk of work with a stored procedure, as opposed to the client executing individual server statements can really reduce network traffic
- **Greater Security** - Some companies only allow developers permission to execute and retrieve database information via stored procedures. This means less chance of mishaps, as direct access to the underlying tables is never granted

Over the next few pages, we're going to cover four common stored procedures, alongside calling Visual Basic code (without error handling for simplicity). In addition, we'll also be demonstrating output parameters.

All of this code can be used with the sample Pubs database that ships with SQL Server. Don't forget to ensure the user ID you use has execution permissions for the stored procedure. Also, remember to add a project reference to the ADO Library before running the Visual Basic code.

To create a Stored Procedure:

- View your Database in Enterprise Manager (as before)

- Right-click on the 'Stored Procedures' icon, selecting 'New Stored Procedure'
- Type your stored procedure statement
- Click OK to Save

To edit a Stored Procedure:

- View your Database in Enterprise Manager (as before)
- Click the 'Stored Procedures' icon
- Double-click on your stored procedure
- Edit as required
 - *You can't change the name of a stored procedure once you've created it. To do this, you'll have to completely recreate it, then delete the original*
- Click OK to Save

SELECT Stored Procedure

Let's look at an example stored procedure that accepts a parameter and returns a set of records to the client.

```
CREATE PROCEDURE SelectTitleByKeyword @keyword varchar(50)
```

```
AS
```

```
SELECT Title, Title_ID, Type
FROM Titles
WHERE Title LIKE '%' + @keyword + '%'
```

This stored procedure is called SelectTitleByKeyword. It accepts a keyword of type varchar(50), a string up to 50 characters long.

It then returns the results of our SQL statement, which searches for any items with the passed keyword in the title.

Visual Basic Code

```
Private Sub Command1_Click()
```

```
Dim objConn As New ADODB.Connection
Dim objCommand As New ADODB.Command
Dim objRS As Recordset
Dim strBookTitle As String
```

```
'Objects to be used in this operation -
'objConn, the basic connection
'objCommand, which handles the stored procedure
'objRS, to hold the returning recordset
'strBookTitle, a string to hold the book keyword
```

```
With objConn
    .ConnectionString = "Driver=SQL Server;Server=COLOSSI;" & _
```

```

        "Database=PUBS;User ID=KarlMoore;Password=TEST"
    .Open
End With

'Get a connection to the database

With objCommand
    .CommandType = adCmdStoredProc
    .CommandText = "SelectTitleByKeyword"
    .ActiveConnection = objConn
End With

'Tell objCommand what it will be working with

strBookTitle = InputBox("Enter a keyword from the book you are looking for", "Book Keyword", "computer")

'Put the keyword into the array. We need to use arrays
'for this, even if we're just working with one argument.

Set objRS = objCommand.Execute(, strBookTitle)

'Execute the stored procedure, passing it the parameter

If objRS.BOF = False And objRS.EOF = False Then
    Call MsgBox("The first book I found containing your " & _
        "keyword was:" & vbCrLf & objRS.Fields("Title") & _
        vbCrLf & "The book ID number is " & _
        objRS.Fields("Title_ID") & vbCrLf & _
        "This book is classified under " & _
        objRS.Fields("Type"), vbInformation)
Else
    MsgBox ("No books contained the keyword: " & strBookTitle)
End If

'Hurrah! We now have a recordset to do what we want with!

Set objRS = Nothing
Set objCommand = Nothing
objConn.Close
Set objConn = Nothing

'Close all references

End Sub

```

UPDATE Stored Procedure

Let's look at an example stored procedure that accepts two parameters and updates records in a table.

```
CREATE PROCEDURE UpdateTitle @OldTitle varchar(80), @NewTitle varchar(80)
```

AS

```
UPDATE Titles
SET Titles.Title = @NewTitle
WHERE Titles.Title = @OldTitle
```

This stored procedure is called UpdateTitle. It accepts two arguments, an OldTitle of type varchar(80) and a NewTitle also of type varchar(80).

In this SQL statement, both the OldTitle and NewTitle arguments refer to the Title field in the Titles table. This is of type varchar(80), hence why the parameters are also of the type varchar(80). You can view the field data types by viewing the properties of a table (as before).

This stored procedure doesn't return anything. It just updates the base table.

Visual Basic Code

```
Private Sub Command1_Click()

Dim objConn As New ADODB.Connection
Dim objCommand As New ADODB.Command
Dim Params(1 To 2) As Variant

'Objects to be used in this operation -
'objConn, the basic connection
'objCommand, which handles the stored procedure
'Params(1 To 2), a variant array to hold the two
' arguments this stored procedures requires
' NOTE: When passing arrays across to stored procedures
' you must declare them as the variant type, unless
' you like the phrase 'major problems'...

With objConn
    .ConnectionString = "Driver=SQL Server;Server=COLOSSI;" & _
        "Database=PUBS;User ID=KarlMoore;Password=TEST"
    .Open
End With

'Get a connection to the database

With objCommand
    .CommandType = adCmdStoredProc
    .CommandText = "UpdateTitle"
    .ActiveConnection = objConn
End With

'Tell objCommand what it will be working with

Params(1) = InputBox("Enter the book title to change:", "Book Title")

Params(2) = InputBox("Enter the new title for " & strParams(1), "New
```

```
'Put the old title and new title into the array,
'ready to pass as arguments to the stored procedure

objCommand.Execute , Params

'Execute the stored procedure, passing it the parameter

Set objCommand = Nothing
objConn.Close
Set objConn = Nothing

'Close all references

End Sub
```

DELETE Stored Procedure

Let's look at an example stored procedure that accepts a numeric ID and deletes all records from the Employee table with the passed Job ID.

```
CREATE PROCEDURE DeleteJobType @JobID smallint
AS

DELETE FROM Employees
WHERE Employees.Job_ID = @JobID
```

This stored procedure is called DeleteJobType. It accepts an ID number of type smallint. It doesn't return anything.

Visual Basic Code

```
Private Sub Command1_Click()

Dim objConn As New ADODB.Connection
Dim objCommand As New ADODB.Command
Dim intJobID As Integer

'Objects to be used in this operation -
'objConn, the basic connection
'objCommand, which handles the stored procedure
'intJobID, an integer to hold the Job IDs to be deleted

With objConn
    .ConnectionString = "Driver=SQL Server;Server=COLOSSI;" & _
        "Database=PUBS;User ID=KarlMoore;Password=TEST"
    .Open
End With

'Get a connection to the database

With objCommand
    .CommandType = adCmdStoredProc
```

```

        .CommandText = "DeleteJobType"
        .ActiveConnection = objConn
End With

'Tell objCommand what it will be working with

intJobID = InputBox("Which job type do you wish to delete?", "Enter

'Grab the Job ID from the user

objCommand.Execute , intJobID

'Execute the stored procedure, passing it the parameter

Set objCommand = Nothing
objConn.Close
Set objConn = Nothing

'Close all references

End Sub

```

INSERT Stored Procedure

Let's look at an example stored procedure that accepts a number of parameters, then uses them to insert a row into a table.

```

CREATE PROCEDURE CreateNewJobType
@job_desc varchar(50), @min_lvl tinyint, @max_lvl tinyint

AS

INSERT INTO jobs (job_desc, min_lvl, max_lvl)
VALUES (@job_desc, @min_lvl, @max_lvl)

```

This stored procedure is called CreateNewJob. It accepts three parameters of various types. It then proceeds to insert a new record into the Jobs table using these arguments.

Visual Basic Code

```

Private Sub Command1_Click()

Dim objConn As New ADODB.Connection
Dim objCommand As New ADODB.Command
Dim Params(1 To 3) As Variant

'Objects to be used in this operation -
'objConn, the basic connection
'objCommand, which handles the stored procedure
'Params(1 To 3), an array to hold the arguments

With objConn

```

```

        .ConnectionString = "Driver=SQL Server;Server=COLOSSI;" & _
            "Database=PUBS;User ID=KarlMoore;Password=TEST"
    .Open
End With

'Get a connection to the database

With objCommand
    .CommandType = adCmdStoredProc
    .CommandText = "CreateNewJobType"
    .ActiveConnection = objConn
End With

'Tell objCommand what it will be working with

Params(1) = InputBox("Enter the job description (varchar(50)):")
Params(2) = InputBox("Enter the minimum level (tinyint):")
Params(3) = InputBox("Enter the maximum level (tinyint):")

'Grab the arguments from the user

objCommand.Execute , Params

'Execute the stored procedure, passing it the parameter

Set objCommand = Nothing
objConn.Close
Set objConn = Nothing

'Close all references

End Sub

```

Output Parameters

Passing information back to your Visual Basic program via stored procedures is commonly done via recordsets. But that isn't always the best solution.

Sometimes you may want to pass your procedure an author ID, and have it return just the author's first and last names. Or perhaps you want a procedure to tell you exactly how many customers you currently have. Maybe you simply want to return a custom error message.

You can do all this via output parameters.

Let's look at an example stored procedure:

```

CREATE PROCEDURE Hello @HelloMsg varchar(50) output

AS

SET @HelloMsg="Good Morning from SQL Server!"

```

RETURN

This procedure is called Hello and accepts a typical varchar parameter called HelloMsg. However note that it's followed by the 'output' keyword.

The procedure then sets the HelloMsg parameter to a simple message, and finishes off cleanly with the RETURN statement (not required but certainly neat, akin to an Exit Function command).

Let's now look at some Visual Basic code that could handle this. On the next page, we'll look at an example both more complicated and useful.

Visual Basic Code

```
Private Sub Command1_Click()

Dim objConn As New ADODB.Connection
Dim objCommand As New ADODB.Command
Dim objParam As Parameter

'Objects to be used in this operation -
'objConn, the basic connection
'objCommand, which handles the stored procedure
'objParam, the parameter we're passing

With objConn
    .ConnectionString = "Driver=SQL Server;Server=COLOSSI;" & _
        "Database=PUBS;User ID=KarlMoore;Password=TEST"
    .Open
End With

'Get a connection to the database

With objCommand
    .CommandType = adCmdStoredProc
    .CommandText = "Hello"
    .ActiveConnection = objConn
End With

'Tell objCommand what it will be working with

Set objParam = objCommand.CreateParameter("TheHelloMessage", _
adVarChar, adParamOutput, 50)

'Create the parameter, stating its data type
'and that it is an output parameter (adParamOutput).
'The name of the parameter doesn't matter, but if
'you're passing more than one, do so in the order
'they're listed in the stored procedure.

objCommand.Parameters.Append objParam

'Add our parameter to objCommand's Parameters collection
```

```
objCommand.Execute

'Execute the stored procedure
'Our parameter will be automatically passed

MsgBox objParam.Value

'After the stored procedure is altered, any return
'value will now be inside objParam.Value

Set objParam = Nothing
Set objCommand = Nothing
objConn.Close
Set objConn = Nothing

'Close all references

End Sub
```

More Output Parameters

Let's now look at another sample to demonstrate output parameters.

Consider this example stored procedure:

```
CREATE PROCEDURE GetAuthorNames
@AuthorID id, @FirstName varchar(20) output, @LastName varchar(40) c

AS

SELECT @FirstName = au_fname, @LastName = au_lname
FROM Authors
WHERE au_id = @AuthorID

RETURN
```

This procedure is called GetAuthorNames and accepts three parameters; an Author ID of type id (varchar), plus two output parameters, FirstName and LastName.

The procedure dips into the Authors database and sets the FirstName and LastName output parameters to the relevant au_fname and au_lname fields in the table.

Let's look at the Visual Basic code that could handle this.

Visual Basic Code

```
Private Sub Command1_Click()

Dim objConn As New ADODB.Connection
```

```
Dim objCommand As New ADODB.Command
Dim objParam1 As Parameter
Dim objParam2 As Parameter
Dim objParam3 As Parameter
Dim strAuthorID As String

'Objects to be used in this operation -
'objConn, the basic connection
'objCommand, which handles the stored procedure
'objParam1/2/3, the parameters we're passing
'strAuthorID, variable to hold requested author ID

With objConn
    .ConnectionString = "Driver=SQL Server;Server=COLOSSI;" & _
        "Database=PUBS;User ID=KarlMoore;Password=TEST"
    .Open
End With

'Get a connection to the database

With objCommand
    .CommandType = adCmdStoredProc
    .CommandText = "GetAuthorNames"
    .ActiveConnection = objConn
End With

'Tell objCommand what it will be working with

strAuthorID = InputBox("Enter the author ID:", "Required", "341-22-1")

'Ask the user for the author ID

Set objParam1 = objCommand.CreateParameter("AuthorID", _
adVarChar, adParamInput, 11, strAuthorID)
Set objParam2 = objCommand.CreateParameter("FirstName", _
adVarChar, adParamOutput, 20)
Set objParam3 = objCommand.CreateParameter("LastName", _
adVarChar, adParamOutput, 40)

'Create the parameters - both input and outputs types
'Note that objParam1 is an input parameter and also
'specifies a passed value, our strAuthorID

objCommand.Parameters.Append objParam1
objCommand.Parameters.Append objParam2
objCommand.Parameters.Append objParam3

'Add our parameter to objCommand's Parameters collection
'You would typically do this straight after you create
'the parameters. Also, you need to append these in the
'order the parameters are listed in the stored procedure

objCommand.Execute
```

```
'Execute the stored procedure
'Our parameters will be automatically passed

If IsNull(objParam2.Value) And IsNull(objParam3.Value) Then
    MsgBox "The return parameters are null. So you " & _
        "probably entered an invalid author ID number!", _
        vbExclamation, "No Such Author ID: " & strAuthorID

Else
    MsgBox "Author ID: " & strAuthorID & vbNewLine & _
        "First Name: " & objParam2.Value & vbNewLine & _
        "Last Name: " & objParam3.Value, vbInformation

End If

'Check the parameters and display the values appropriately

Set objParam = Nothing
Set objCommand = Nothing
objConn.Close
Set objConn = Nothing

'Close all references

End Sub
```

That's the end of our quick start guide to SQL Server 7. I hope you've enjoyed the series. Visit us again soon at www.VBWorld.com for even more quick start guides!