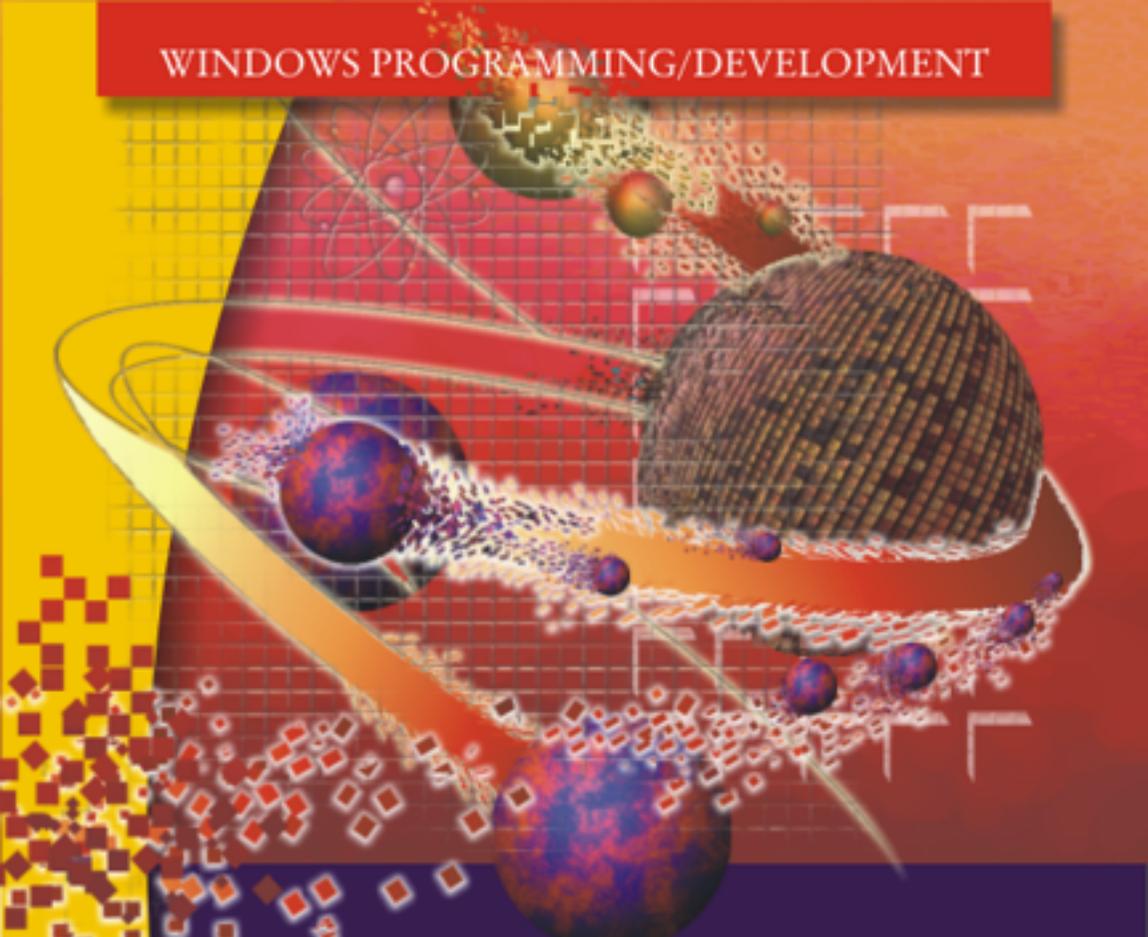


WINDOWS PROGRAMMING/DEVELOPMENT



ADO .NET

Programming



Companion
CD-ROM
Included



***Terrence J. Joubert
and Ryan N. Payet***

TEAM LinG - Live, Informative, Non-cost and Genuine!

ADO .NET Programming

**Terrence J. Joubert
and Ryan N. Payet**

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Joubert, Terrence J.

ADO .NET programming / by Terrence J. Joubert and Ryan N. Payet.

p. cm.

Includes index.

ISBN 1-55622-965-8 (paperback)

1. Internet programming. 2. ActiveX. 3. Microsoft .NET. I. Payet, Ryan N. II. Title.

QA76.625 J69 2002

005.2'76--dc21

200212695

CIP

© 2003, Wordware Publishing, Inc.

All Rights Reserved

2320 Los Rios Boulevard

Plano, Texas 75074

No part of this book may be reproduced in any form or by
any means without permission in writing from
Wordware Publishing, Inc.

Printed in the United States of America

ISBN 1-55622-965-8

10 9 8 7 6 5 4 3 2 1

0210

Products mentioned are used for identification purposes only and may be trademarks
of their respective companies.

All inquiries for volume purchases of this book should be addressed to
Wordware Publishing, Inc., at the above address. Telephone inquiries may be
made by calling:

(972) 423-0090

TEAM LinG - Live, Informative, Non-cost and Genuine!

Contents

Aims and Objectives.	xiii
Part I: Introduction to ADO .NET	1
Chapter 1: Growing up from ADO.	3
In This Chapter	3
Architectural Differences.	3
The ADO Architecture	4
Why Not Use ADO in .NET?.	5
The ADO .NET Architecture	7
.NET Data Providers	9
In-Memory Data Representation	10
ADO: The Recordset Object	10
ADO .NET: The DataSet and DataTable Objects	10
Relationship Management	12
ADO: Using JOIN in SQL	12
ADO .NET: The DataRelation Object	13
Where is the Recordset?	15
Summary.	16
Part II: ADO .NET Revealed	17
Chapter 2: Interacting with Databases.	19
In This Chapter	19
The Connection Object	20
Connection Object Properties	21
ConnectionString	21
ConnectionTimeout	22
Database	22
DataSource.	23

Contents

Provider	23
ServerVersion	23
State	24
Connection Object Methods	24
BeginTransaction()	24
ChangeDatabase()	25
Close()	25
CreateCommand()	26
Dispose()	26
Equals()	27
GetType()	27
Open()	28
ToString()	28
Connecting Through SQL Server .NET	
Data Provider	29
The Command Object	30
Command Object Properties	30
CommandText	30
CommandTimeout	30
CommandType	31
Connection	31
Container	32
Parameters	32
Transaction	32
Command Object Methods	33
Cancel()	33
CreateParameter()	33
Dispose()	34
ExecuteNonQuery()	34
ExecuteReader()	35
ExecuteScalar()	35
ExecuteXmlReader()	36
GetType()	36
ToString()	37
The DataReader Object	37
DataReader Properties	37
Depth	37

FieldCount	38
IsClosed	38
Item(<column_name as string> or <column_ordinal as integer>)	38
RecordsAffected	39
DataReader Methods	39
Close()	39
CreateObjRef().	39
Equals()	40
GetBoolean().	40
GetByte()	41
GetBytes().	41
GetChar()	42
GetChars().	43
GetDataTypeName().	43
GetDateTime().	44
GetDecimal()	44
GetDouble()	45
GetFieldType().	45
GetFloat()	46
GetGuid()	46
GetHashCode()	47
GetInt16()	47
GetInt32()	48
GetInt64()	49
GetName().	49
GetOrdinal()	50
GetSchemaTable().	50
GetString().	50
GetTimeSpan()	51
GetValue()	51
GetValues()	52
IsDBNull()	52
NextResult().	53
Read()	53
The DataAdapter Object	54
SqlDataAdapter Constructor.	54

Contents

New().	55
New(System.Data.SqlClient.SqlCommand)	55
New(command as String, connection as String)	56
New(command as String, connection as SqlConnection)	56
SqlDataAdapter Properties.	57
AcceptChangesDuringFill	57
ContinueUpdateOnError	58
DeleteCommand.	58
InsertCommand	59
MissingMappingAction	59
MissingSchemaAction.	60
SelectCommand	60
TableMappings.	61
UpdateCommand	61
SqlDataAdapter Methods.	62
CreateObjRef().	62
Dispose().	62
Fill()	63
FillSchema()	64
Update()	65
Summary.	66
Chapter 3: Data Manipulation	67
In This Chapter	67
The DataSet Component	68
What is the DataSet?	68
When Do You Need the DataSet?	69
How is the DataSet Organized?	70
Core DataSet Properties	71
CaseSensitive	71
Container	71
DataSetName	72
DefaultViewManager	72
DesignMode	72
EnforceConstraints	73

ExtendedProperties	73
HasErrors	73
Locale	74
Namespace.	74
Prefix.	74
Relations	75
Site	75
Tables	75
Core DataSet Methods	76
AcceptChanges().	76
BeginInit()	76
Clear()	77
Clone()	78
Copy()	78
Dispose().	79
EndInit()	79
GetChanges()	80
GetService().	80
GetType()	81
GetXML()	82
GetXmlSchema()	82
HasChanges()	83
InferXmlSchema().	84
Merge()	86
ReadXml()	88
ReadXmlSchema().	89
RejectChanges().	89
Reset()	90
ToString()	90
WriteXml().	91
WriteXmlSchema()	91
DataSet.ExtendedProperties	92
Adding an Extended Property	92
Reading and Writing Values	93
DataTableCollection.	93
DataRelationCollection	94
The Big Picture	95

Contents

Summary.	98
Chapter 4: Designing ADO .NET Applications	99
.NET Application Models	99
Windows Forms Applications	99
Form Data Binding	100
Common Scenarios for Data Binding	102
Data Access Strategy for Windows Forms Applications.	103
Console Applications	104
Data Access Strategy for Console Applications.	104
Windows Services Applications	105
Data Access Strategy for Windows Services . . .	105
ASP .NET Web Applications	106
Web Forms	107
Data Access in Web Forms.	107
Data to XML Web Services	109
Data Access Strategy for ASP .NET Applications.	115
Data, Data Everywhere.	120
Spec My Components	120
What is a Component in .NET?.	121
When to Build Data Components.	121
Component Design Guidelines	122
Component Implementation	123
Learning to Run	125
Connection Pooling	125
Stored Procedure or SQL Statement?	127
Which Data Type?.	127
Data Warehousing.	128
Tuning and Monitoring	128
Protecting the Application.	129
Passwords, Users, and Access Rights	129
Application Information.	130
Summary	130

Chapter 5: XML Integration with ADO .NET	131
XML in .NET Frameworks	131
Architectural Overview and Design Goals.	131
Standards Compliance	132
Extensibility	132
Pluggable Architecture.	132
Performance	133
Tight Integration with ADO .NET.	133
DOM: The XML Document Object Model.	134
Nodes in .NET	137
Loading XML Documents in the DOM	138
Validating XML Documents	139
XML Integration with Relational Data	143
XML with MS SQL Server 2000	143
DataSet and XML	144
DiffGrams.	145
Working with ReadXml	149
Writing XML from DataSet	152
XML Schemas from DataSet.	154
Typed DataSets from XSD Schema	155
DataSet and XmlDataDocument	156
Synchronizing DataSet with XmlDataDocument.	157
Nested DataRelations	159
Creating DataSet Relational Schema from XML Schema	164
Creating from XML Schema (XSD)	164
Inferring from XML	170
Summary	172
Chapter 6: Practical ADO .NET Programming (Part One)	173
In This Chapter	173
The Case Study	174
The Web Service.	175

Contents

Designing the Web Service	175
OrderProcessingWS	176
Data Retrieval Methods	176
Implementing OrderProcessingWS	177
Setting Up IIS	178
Creating OrderProcessingWS Project	179
Web Service Namespace	180
Initialization Code	181
GetOrders Methods	184
GetOrderDetails Methods	193
GetFullOrders Methods	195
GetFullOrders Code	201
GetFullOrders_By_Customer Code	205
Summary	207
Chapter 7: Practical ADO .NET Programming (Part Two)	209
In This Chapter	209
Data Update Methods	209
The Update Functions	210
The Protected Order Details Update Methods	211
The DeleteOrderDetails Method	211
The InsertOrderDetails Method	214
The UpdateOrderDetails Method	216
Concurrency Issues	219
The Protected Orders Update Methods	220
The sp_UpdateOrders Stored Procedure	220
The sp_InsertOrders Stored Procedure	223
The sp_DeleteOrders Stored Procedure	227
The UpdateOrders Method	228
The DeleteOrders Method	232
The InsertOrders Method	234
The FullUpdateOrder Method	237
Testing the Update Methods	241
Summary	241

Part III: Special Topics 243

Chapter 8: Migrating ADO Applications 245

 In This Chapter 245

 Legacy of Time. 245

 Language Changes. 246

 What about COM? 249

 .NET Framework Bidirectional Migration Support 250

 ASP and ASP .NET 250

 What about ADO? 250

 To Migrate or Not to Migrate? 257

 Migration Steps 258

 Step 1: Migrate the Clients 258

 Step 2: Create .NET Wrappers to COM Components 258

 Step 3: Migrate the Business Objects 259

 Summary 259

Chapter 9: Manipulating Multidimensional Data. 261

 In This Chapter 261

 A Quick Primer on Analysis Services. 262

 Analysis Services Installation 262

 System Requirements 262

 Installation Components 264

 Setup 266

 Starting Up 266

 Running Setup. 266

 Understanding the Data Source 270

 The Relational Database 271

 The OLAP Database 272

 Populating the OLAP Database 272

 How is the Data Stored? 273

 PivotTable Service 273

 OLEDB Provider for OLAP 273

 Multidimensional Expressions (MDX) 274

Contents

ActiveX Data Object Multidimensional (ADO MD)	274
ADO MD Example	284
Using the CubeBrowser ActiveX Control	285
Summary	297
Appendix A: The Object-Oriented Features of VB .NET	299
Appendix B: Database Normalization.	325
Appendix C: Views, Stored Procedures, and Triggers . .	335
Appendix D: Advanced SQL Query Techniques	375
Index.	417

Aims and Objectives

This book provides a sophisticated reference to ADO .NET solution development using Microsoft Visual Studio .NET. It is aimed at programmers with a working knowledge of the .NET Framework and VB .NET. A beginner's knowledge of ADO .NET is not necessary, but it will provide an advantage. Much of the ADO .NET functionality is specifically targeted at developers, and the aim of this book is to dive into the advanced topics and various programming opportunities that the product presents.

The book will assume readers have experience and familiarity with the following technologies:

- OLE DB data access technologies
- The .NET Framework
- ADO/ADO .NET
- XML (Extensible Markup Language)
- Visual Studio .NET

The book takes a specifically solutions-oriented approach, demonstrating at all levels how the product can be used to provide timely solutions to real-world problems. Similarly, an emphasis will be placed on the process of solution development using robust examples to teach how concepts are applied in the business world.

Many readers will read the book in sequence, from cover to cover, in order to get up to speed and become familiar with the product as quickly as possible. Others will wish

Aims and Objectives

to dip in on individual chapters, even individual sections, effectively using the book as a reference volume. I aim to cater as much as possible to both groups; each chapter has clearly defined content that builds on previously discussed material but does not, in any way, rely on the material that follows it.

The content develops in a consistent, logical manner, which advances the “story” of the book; that is, just as the book as a whole has a direction and purpose, each chapter, even individual sections within a chapter, have a well-defined direction and purpose.

Part I

Introduction to ADO .NET

Chapter 1

Growing up from ADO

In This Chapter

ADO .NET presents a robust and revolutionary data access architecture at the core of the Microsoft .NET strategy. Being an integral member of the core class libraries of the .NET Framework, ADO .NET is nothing like its predecessor. While it does provide some traditional interfaces for backward compatibility and ADO migration, the rich set of tools available in the System.Data namespace that holds ADO .NET goes far beyond the most wonderful magic one can perform with ADO.

This chapter is about the differences between a father and a son. It is important for you, the ADO programmer, to understand the differences at the core conceptual level before you attempt to dive into anything that involves ADO .NET.

Architectural Differences

Architecture is everything. It affects how entities are designed, how they work, and, ultimately, it defines what we can do with them. ADO and ADO .NET have two completely different architectures. This section of the chapter touches on the fundamental principles behind the two technologies and how they differ.

The ADO Architecture

Turn the pages of your history book. Microsoft released ADO as part of its Universal Data Access (UDA) strategy. OLE DB was also released as part of UDA and was originally the standard that developers would use to access and manipulate data sources. Contrary to the ODBC standard that was tied to Windows, OLE DB brought some world peace to the data access industry by allowing database engine developers to implement providers that would essentially provide OLE DB interfaces to data sources. A key example is the Microsoft SQL Server database engine. SQL Server data sources are stored on a database server. The SQLOLE DB provider provides all the means of communication that any client or other servers need with that particular data source. An OLE DB provider may even allow developers to use customized query languages in querying the data sources. In the case of SQLOLE DB, that language is called T-SQL.

To implement an OLE DB provider, one would use the OLE DB API. This is a set of low-level functions written in the C language. With the implementation of such an open architecture, Microsoft was clearly targeting the big database vendors, such as Oracle, Informix, and Sybase, while creating unified data access architecture for its flagship operating system—Windows.

To facilitate rapid development of database applications and further expand its COM (Component Object Modeling) standard, Microsoft released ADO as a library of rich COM components providing high-level interfaces that would sit one layer above the very hard-to-use OLE DB API. Using ADO, database developers would simply attempt a connection to a data source, query the data source, and load a set of rows (record set) into memory.

Why Not Use ADO in .NET?

The .NET Framework does provide support for ADO. In fact, it is possible to create a fully functional ADO application that runs on the .NET Framework. However, if you intend to create an ADO application, I suggest that you stick to Visual Studio 6 because there are a lot of architectural issues that you will face.

1. ADO is Based on COM

As mentioned above, the ADO architecture is based on the principles of COM. The .NET Framework does not support COM directly, but rather, it has a COM interoperability class library that sleeps silently within the COMLIB namespace. When using COM in .NET applications, the CLR always sends requests to the COM interoperability layer whenever you reference anything that is based on COM. This is always the case when you reference ADO objects in a .NET application. It has an enormous negative impact on the speed at which such an application runs. There is no need to pay such a high cost when you have the ADO .NET class library to use that is directly executable by the CLR.

2. Data Type Compatibility

Secondly, there is an issue with the famous ADO Recordset object. This object is used to retrieve and store data in memory. It has a property of the COM type variant called Fields. The .NET CLR does not support the variant data type, as is the case with several other COM data types. Whenever ADO is used in the .NET application, there are a series of data type conversions that the CLR needs to do through the COM interoperability library. This again puts a huge burden on the application's performance.

For a list of COM data types found in ADO and their corresponding .NET data types, refer to the table below:

ADO Data Type	.NET Framework Data Type
adEmpty	null
adBoolean	Int16
adTinyInt	SByte
adSmallInt	Int16
adInteger	Int32
adBigInt	Int64
adUnsignedTinyInt	Int16
adUnsignedSmallInt	Int32
adUnsignedInt	Int64
adUnsignedBigInt	Decimal
adSingle	Single
adDouble	Double
adCurrency	Decimal
adDecimal	Decimal
adNumeric	Decimal
adDate	DateTime
adDBDate	DateTime
adDBTime	DateTime
adDBTimeStamp	DateTime
adFileTime	DateTime
adGUID	GUID
adError	ExternalException
adIUnknown	Object
adIDispatch	Object
adVariant	Object
adPropVariant	Object
adBinary	byte
adChar	string

ADO Data Type	.NET Framework Data Type
adWChar	string
adBSTR	string
adChapter	not supported by .NET
adUserDefined	not supported by .NET
adVarNumeric	not supported by .NET

3. .NET Application Architecture

Third, there are also the design goals of the .NET Framework to consider. .NET was designed so that application services could talk to each other using technologies such as XML and remoting. ADO does not support XML, nor does it allow disconnected data sharing between remote application servers. Using ADO in .NET applications limits the amount of things that you can do to implement a true .NET application.

The ADO .NET Architecture

A great part of the .NET Framework is called *.NET class libraries*. These libraries contain a set of classes that provide services that developers use to create .NET applications. Among these libraries is the System.Data class library.

System.Data holds all the functionality needed for developers to create rich, scalable, and distributed database applications that run on the .NET Common Language Runtime. System.Data provides a unified, object-oriented, hierarchical, and extensible set of classes. It is a common API that is usable across all .NET programming languages, while the CLR enables cross-language inheritance, error handling, and debugging of data classes. Together, the functionality present in these classes is referred to as ADO .NET, and this is what we will discuss throughout the rest of this book.

There are three main design goals behind ADO .NET:

- To provide seamless support for XML
- To provide an expandable and scalable data access architecture for the revolutionary n-tier programming model
- To extend the current capabilities of ADO

The core ADO .NET architecture is built to provide access to relational, XML, and other OLE DB data sources. The XML-centric classes are not within the System.Data library; they are part of the System.XML library. Therefore, it is important to reference both the System.Data and System.XML namespaces in any database applications that you create. In contrast to the ADO object model, ADO .NET separates data access from data manipulation, therefore enabling flexibility when operating in a disconnected environment.

In a typical data manipulation scenario, data is retrieved using the DataReader object through a .NET data provider. The retrieved data can be processed directly or, alternatively, it can be stored inside a DataSet object on the client. Within the DataSet, the data is completely separated from its source. The DataSet provides the developer with three powerful and flexible aspects:

- Data within the DataSet object can be manipulated and exposed to the user in several ways using other objects like DataTable, DataRelation, and DataViews contained within that DataSet.
- The content of a DataSet can be combined with data from other data sources. This offers the developer the superb capability to unify several types of data. Think of a scenario where you may have to implement a link between your Sales, CRM, and Accounting systems. Traditionally, this would have been done using classical COM interfaces. The ADO .NET

DataSet hides all these complexities by providing seamless data integration capability.

- The DataSet provides the ability for you to share data among remote tiers of an application or application servers. Although many developers prefer to share data using XML, this is an excellent way to share data between tiers in the n-tier application programming model.

More is written about the DataSet object in Chapter 3, “Data Manipulation.”

.NET Data Providers

In a typical .NET database application, objects and data-sharing services, referred to as *consumers*, interact with the database using a .NET data provider. To further isolate data access from data manipulation, a provider is implemented to act as a form of interface between data consumers and data servers. It holds all the logic required to perform, retrieve, and update operations on a data source. The provider would be used to retrieve data into a DataSet and update the database.

ADO .NET ships with two data providers: the SQL Server .NET Data Provider and the OLE DB .NET Data Provider.

► **Note:** Other independent database vendors such as Oracle, Informix, and Sybase, may implement data providers for their own data sources.

The SQL Server .NET Data Provider is the provider used for SQL Server databases implemented in SQL Server 7 or later. Data access to OLE DB compliant data sources is achieved using the OLE DB .NET Data Provider.

The remainder of this chapter covers the differences between ADO and ADO .NET.

In-Memory Data Representation

Manipulating data in a client application is one of the most important aspects of database programming. In order to effectively implement high-performance data manipulation procedures, it is imperative for you to understand how the data access components hold data in memory.

ADO: The Recordset Object

The way data resides inside an ADO Recordset object is somewhat similar to database tables. The data is represented in single rows. A value inside a column is accessed by browsing to the record and reading the value attribute of the Recordset's Fields(columnName) object property. A reading of a Recordset value is illustrated here:

```
Set objRS = Server.CreateObject("ADODB.Recordset")
'retrieve a set of rows
Variable1 = objRs.Fields("FieldName")
```

ADO .NET: The DataSet and DataTable Objects

The DataSet object holds data values in memory in ADO .NET applications. The DataSet itself is not a direct container of data. It holds a collection of Recordset-like objects called *DataTables*. A DataTable is the client representation of a database table; it is simply a set of rows. The DataSet and DataTable objects are covered in more detail in Chapter 3, “Data Manipulation.”

The following code illustrates the way in which a DataSet is initialized on a Web Form:

```

Dim objDSArtists As New _
    System.Data.DataSet("Artists")

Dim objDAArtists As _
    System.Data.SqlClient.SqlDataAdapter

Dim strQuery, _
    strCon, _
    strArtistName _
    As String

Dim intArtistID As Long

Dim intLoopCounter As Integer

strCon = "Initial Catalog=Multimedia;"
strCon &= "Data Source=LOCALHOST;UID=sa;"
strCon &= "PWD=sa;"

strQuery = "SELECT * FROM Artist "
strQuery &= "Order By ArtistName"

objDAArtists = New _
    SqlConnection.SqlDataAdapter(strQuery, strCon)

objDAArtists.Fill(objDSArtists)

lstSingers.Items.Clear()

With objDSArtists.Tables(0)
    For intLoopCounter = 0 To .Rows.Count - 1
        strArtistName = _
            .Rows(intLoopCounter).Item("ArtistName")
        lstSingers.Items.Add(strArtistName)
    Next
End With

```

1. The DataSet is initialized followed by a DataAdapter object from the SQL Server .NET data provider, SqlConnection.
2. Other variables that would be used for DataSet manipulation are also declared and initialized.

3. The DataAdapter is then initialized with a string representing the query and the connection string for the database on the SQL Server.
4. lstSingers is a ListBox web control. The purpose of the code above is to retrieve a list of Singers and populate the ListBox.
5. After data retrieval, the DataSet is populated by whatever is retrieved by the DataAdapter object, using the latter's Fill method. When the Fill method is called, the DataSet, which is passed as a parameter, does the following:
 - a. Internally creates a DataTable object
 - b. Initializes that particular DataTable with the row set retrieved by the DataAdapter
6. The Tables collection property of the DataSet holds references to all the DataTable objects that are part of the particular DataSet. Each object inside the collection has an index. Since the code above created only one DataTable, we are safe in referencing that DataTable with the index 0; otherwise, it would have been better to reference it using a particular name specified when the Fill method is called on the DataAdapter.

Relationship Management

The maintenance of relationships between data elements is an imperative part of data manipulation inside client applications. Let's take a look at how ADO and ADO .NET differ in the way they maintain relationships between related data elements.

ADO: Using JOIN in SQL

In order to maintain proper relationships between data rows in ADO Recordset objects, the JOIN SQL statement

is used to retrieve the data. A typical relationship scenario would be a query that retrieves the name and address details of all customers that ordered products during the first quarter of 1997. In ADO, this operation is handled using the following code:

```
strQuery = "SELECT DISTINCT Customers.CompanyName,
           Customers.City, Customers.Country "
strQuery = strQuery & "FROM Customers RIGHT JOIN Orders
           ON Customers.CustomerID = Orders.CustomerID"
strQuery = strQuery & "WHERE Orders.OrderDate BETWEEN
           '19970101' And '19970331'"

objRS.Open(strQuery, objConn)    'assuming objConn is a
                                'valid connection object
```

The Recordset would return the following set of rows:

	CompanyName	City	Country
1	Around the Horn	London	UK
2	Berglunds snabbköp	Luleå	Sweden
3	Blondesddsl père et fils	Strasbourg	France
4	Bon app'	Marseille	France
5	Bottom-Dollar Markets	Tsawassen	Canada
6	B's Beverages	London	UK
7	Comércio Mineiro	Sao Paulo	Brazil
8	Consolidated Holdings	London	UK
9	Eastern Connection	London	UK
10	Ernst Handel	Graz	Austria
11	Familia Arquibaldo	Sao Paulo	Brazil
12	Folies gourmandes	Lille	France
13	Folk och få HB	Bräcke	Sweden
14	Franchi S.p.À.	Torino	Italy
15	Frankenversand	München	Germany
16	Furia Bacalhau e Frutos do Mar	Lisboa	Portugal
17	Galería del gastrónomo	Barcelona	Spain
18	Gourmet Lanchonetes	Campinas	Brazil
19	HILARION-Abastos	San Cristóbal	Venezuela
20	Hungry Coyote Import Store	Elgin	USA
21	Hungry Owl All-Night Grocers	Cork	Ireland

ADO .NET: The DataRelation Object

One of the core design goals of ADO .NET was to provide data access in a mobile, web-based, disconnected, and server-isolated application architecture. Implementing JOIN statements like in ADO would mean that the server needs to get involved in every data

manipulation scenario performed by the client. This is nowhere near server isolation.

In ADO .NET, relationships between associated rows of different DataTables are maintained by objects called DataRelations. Refer to Chapter 3 for more information on the DataRelation object. In the following piece of code, we create a DataRelation object for two DataTable objects inside a DataSet:

```
Private Sub Create_Cust_Order_Relation()  
  
    Dim custCol As DataColumn  
    Dim orderCol As DataColumn  
  
    custCol = objDataSet.Tables("Customers")  
                .Columns("CustID")  
    orderCol = objDataSet.Tables("Orders")  
                .Columns("CustID")  
  
    ' Create DataRelation.  
    Dim relCustOrder As DataRelation  
    relCustOrder = New DataRelation("CustomersOrders",  
        custCol, orderCol)  
  
    ' Add the relation to the DataSet.  
    objDataSet.Relations.Add(relCustOrder)  
End Sub
```

As illustrated in the code above, a DataRelation object is created along two DataColumn objects. The DataColumn objects must be of the same data type. Once a DataRelation is created, it is possible to show the data in several dimensions using DataViews.

You will learn more about the DataColumn and DataView objects in Chapter 3.

Where is the Recordset?

Finally, it is very important for ADO programmers to understand where their favorite Recordset object has gone. As the following table illustrates, the functionality provided by the Recordset object is actually shared among several ADO .NET objects.

ADO .NET Object	Description
DataReader	Provides a forward-only and read-only row set of data from a data source. The DataReader is similar to a Recordset object with its CursorType property set to adOpenForwardOnly and its LockType property set to adLockReadOnly.
DataSet	Provides client access to relational data. This object is independent of any specific data source and therefore can be populated from multiple and differing data sources, including relational databases and XML, or can be populated with data local to the application. Data is stored in a collection of one or more table-like structures called DataTables and can be accessed non-sequentially and without limits to availability, unlike ADO in which data must be accessed a single row at a time. A DataSet can contain relationships between tables, similar to the ADO Recordset in which a single result set is created from a JOIN. A DataSet can also contain unique, primary key, and foreign key constraints on its tables. The DataSet is similar to a Recordset with CursorLocation = adUseClient, CursorType = adOpenStatic, and LockType = adLockOptimistic. However, the DataSet has extended capabilities over the Recordset for managing application data.
DataAdapter	Populates a DataSet with data from a relational database and resolves changes in the DataSet back to the data source. The DataAdapter enables you to explicitly specify behavior that the Recordset performs implicitly.

Summary

You have just read a high-level overview of the differences between ADO and ADO .NET. While the .NET Framework provides support for the COM standard on which ADO is built, ADO .NET provides all the means of data access in .NET for all types of application services. ADO .NET is also more integrated with XML than ADO, making data sharing across remote application tiers a reality. ADO .NET contains a lot of objects, and the subsequent chapters dig deeper into the way these objects interoperate to help you implement the best .NET database application.

Part II

ADO .NET Revealed

Chapter 2

Interacting with Databases

In This Chapter

Throughout Chapter 1, you learned about the existence of the components available as two distinct groups in ADO .NET. There is the group of components that allows connection to and interaction with data sources and another group that provides client manipulation of data.

This chapter is about the first aforementioned group of components that provides an interface to the data source. A complete reference is provided on the following components:

- Connection
- Command
- DataReader
- DataAdapter

As covered in Chapter 1, all .NET data providers must implement interfaces to all these components. Figure 2-1 shows the relationship of these components inside a .NET data provider. In this chapter, we will cover each of these components as they are implemented in the two original data providers shipped with the .NET Framework (SQL Server .NET Data Provider and OLE DB .NET Data Provider).

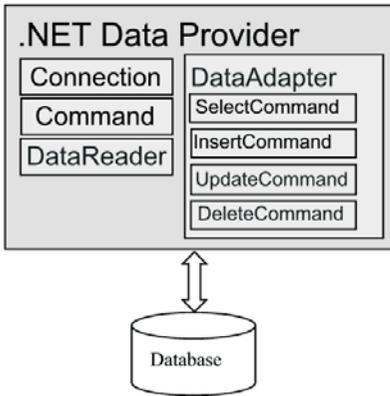


Figure 2-1: A view inside a .NET data provider

This chapter is very conceptual and organized as a reference volume to the above objects. After learning about the concepts in this chapter, you will be provided with an overview of how everything fits together in a practical programming environment in Chapter 5.

The Connection Object

The Connection object provides all the means of connection to the database on which a desired transaction is to be made. This is the first object that any developer comes face-to-face with while trying to use the .NET Data Provider. Before any commands can be executed on the database, a successful connection to that particular database must be established using the Connection object.

Connection Object Properties

ConnectionString

Type: String

Attribute: Read/Write

Default: ""

Description: The ConnectionString property of the Connection object defines a valid connection string to the database with which the Connection object is to connect. The string usually contains parameters necessary to connect to the data source. The string is usually a semicolon-separated parameter list, with each parameter set to its appropriate value:

Provider: Specifies the OLE DB provider through which you wish to connect to a data source. You need to specify a provider only when you wish to connect through the OLE DB .NET Data Provider. This parameter does not have to be specified when connecting through the SQL Server .NET Data Provider.

Data Source: Specifies the name of the database server that is hosting the database to which you wish to connect

Initial Catalog: Specifies the name of the database to which you wish to connect. The database must be located on the Data Source database server.

UserID: Specifies a valid user ID for a user who has access to login to the database server specified by the Data Source parameter

Password: Specifies the password for the user specified in the UserID parameter

A typical connection string would look like this:

```
Connection.ConnectionString = "Provider=MSOLAP;DataSource=LOCALHOST;UserID=sa;password=sa"
```

It is good coding practice to prepare the value of the connection string in one place. A typical scenario is to store the value inside a string variable and then use that variable to set the value of this property.

ConnectionTimeout

Type: Integer

Attribute: Read-only

Default: 15

Description: The ConnectionTimeout property is an integer value that specifies the number of seconds that the Connection object should wait for a connection to be established to a server before generating an error.

Database

Type: String

Attribute: Read-only

Default: ""

Description: The Database property obtains the name of the database to which the Connection object is connected.

A typical use of the Database property is when you wish to show a user which database is currently being used.

DataSource

Type: String

Attribute: Read Only

Default: ""

Description: The DataSource property obtains the location and filename of the data source to which the Connection object is currently connected.

A typical use of the DataSource property is when you wish to show a user which data source is currently being used.

Provider

Type: String

Attribute: Read-only

Default: ""

Description: The Provider property obtains the current OLE DB provider through which the Connection object is connecting to the current data source.

A typical use of the Provider property is when you wish to show a user which provider is currently being used for transactions to a particular data source.

ServerVersion

Type: String

Attribute: Read-only

Default: ""

Description: The ServerVersion property obtains the version information of the database server to which the client is connected.

A typical use of the `ServerVersion` property is when you wish to show a user the version information for the particular server to which the `Connection` object is connected.

State

Type: `ConnectionState` enumeration residing inside the `System.Data` namespace

Attribute: Read-only

Description: The `State` property obtains the current state of the connection to the database server. Possible values are:

Open: The connection to the server exists and can be used to issue commands to the database.

Closed: The connection to the server does not exist and cannot be used to issue commands to the database.

A typical use of the `State` property is when you wish to test whether a valid connection exists to the data source before performing any operations. This is a solid mechanism to avoid errors in your applications.

Connection Object Methods

BeginTransaction()

Returns

System.Data.IDbTransaction: An object that represents a valid database transaction

Parameters

None

Description

Begins a new database transaction

Usage

Call this method to perform any transaction on a database. You must call the Commit() or Rollback() method after the transaction has been made.

ChangeDatabase()**Returns**

Void

Parameters

value: String; the name of the new database

Description

Changes the database for an opened connection. The database must reside on the same server as the previous database.

Usage

This method is used when you wish to change the database that you are working with. It is a very convenient way to manage memory because you are not creating a new connection.

Close()**Returns**

Void

Parameters

None

Description

Closes the connection that the Connection object has with a data source

Usage

This method is used when you wish to close an opened connection.

CreateCommand()**Returns**

System.Data.IDbCommand: A valid Command object

Parameters

None

Description

Creates a valid Command object associated with the Connection object

Usage

This method is used when you wish to create a new Command object. After creating the Command object, you can start issuing SQL commands to the database to which the Connection object has an open connection.

Dispose()**Returns**

Void

Parameters

None

Description

Closes the Connection object and releases all of its resources

Usage

Use the Dispose() method to release memory occupied by the Connection object.

Equals()**Returns**

Boolean

Parameters

obj: System.Object; any object inherited from the System.Object class

Description

Determines whether obj is equal to the Connection object

Usage

This method will always return False if you pass an object that is not of the same type as the Connection object. It is useful when you need to test whether two Connection objects have the same type of connection.

GetType()**Returns**

System.Type

Parameters

None

Description

This method returns the System.Type value for the Connection object. It is usually the class name of the object.

Usage

Use this method to obtain a valid System.Type for the Connection component.

Open()**Returns**

Void

Parameters

None

Description

Opens a database connection based on the settings inside the ConnectionString property

Usage

Use this method to open a connection to the database.

ToString()**Returns**

String: A valid string representing the object

Parameters

None

Description

Attempts to convert the Connection object to a string and returns the string

Usage

Use this method to obtain a string representation of the Connection component.

Connecting Through SQL Server .NET Data Provider

In this section you will learn to connect to an SQL Server database using the SQL Server .NET Data Provider. This is simply to have a taste of how this provider implements its flavor of the Connection object differently from the generic Connection object.

The following code assumes that you are referencing the System.Data.SqlClient namespace that holds all the logic for the SQL Server .NET Data Provider.

- The Connection object and connection string are declared:

```
Dim objConn As New System.Data.SqlClient.  
    SqlConnection()  
Dim strConn As String
```

- The connection string is initialized. Notice that all you have to specify in the connection string is the server name, the initial catalog, and the login information.

```
strConn = "Data Source=LOCALHOST; Initial  
    Catalog=Northwind; UserID=sa;Password=sa"
```

- In the last section of the code, the ConnectionString property of the Connection object is set to the value of strConn, and the object attempts to open a connection to the database server by invoking its Open() method.

```
With objConn  
    .ConnectionString = strConn  
    .Open()  
End With
```

These are all the steps that are required to open a connection to an SQL Server database. The process is the same for other OLE DB data sources, except that you

have to specify an OLE DB provider as the Provider parameter in the connection string.

The Command Object

The Command object is used to issue SQL commands to the database. Although that means any type of SQL command, ADO .NET was optimized to handle SQL commands differently. When issuing DDL (Data Definition Language) commands or invoking a stored procedure, you are safe using the Command object. DML (Data Manipulation Language) commands are usually handled by the objects inside ADO .NET's DataAdapter component. This section is a complete reference to the generic Command object in ADO .NET.

Command Object Properties

CommandText

Type: String

Attribute: Read/Write

Default: ""

Description: The CommandText property gets or sets the SQL command that is to be executed against the data source. If you wish to execute a stored procedure, simply assigning the name of the stored procedure to this property is enough to define a command.

CommandTimeout

Type: Integer

Attribute: Read/Write

Default: 30

Description: The `CommandTimeout` property is an integer value that specifies the number of seconds the `Command` object should wait for a command to be executed against the database before generating an error.

CommandType

Type: `System.Data.CommandType` enumeration

Attribute: Read/Write

Default: Text

Possible values:

Stored Procedure: Interprets the `CommandText` property as a call to execute a stored procedure in the database

TableDirect: Interprets the `CommandText` property as the name of a table inside the database. When the `Command` object is executed, the entire table is retrieved—all its data, plus its schema!

If you intend to retrieve more than one table, use a comma-delimited list of tables without spaces as the `CommandText` property. All the tables and their schema are retrieved.

Text: Interprets the `CommandText` property as an SQL command

Description: The `CommandType` property gets or sets the manner through which the `Command` object will execute its `CommandText` property against the data source.

Connection

Type: `System.Data.[.NET Data Provider].Connection`

Attribute: Read/Write

Default: Null

Description: This property defines a valid ADO .NET Connection object that has an open connection to the data source against which you want to execute the command.

Container

Type: System.ComponentModel.IContainer

Attribute: Read/Write

Default: Null

Description: Defines the component service of which the Command object is a member.

Parameters

Type: System.Data.[*NET Data Provider*].ParameterCollection

Attribute: Read-only

Default: Null

Description: This property returns the collection of items being passed to the command as parameters. A parameter is set using the CreateParameter() method.

Transaction

Type: System.Data.[*NET Data Provider*].ParameterCollection

Attribute: Read/Write

Default: Null

Description: This property gets or sets the valid Transaction object during which this command will execute.

Command Object Methods

Cancel()

Returns

Void

Parameters

None

Description

This method attempts to cancel a command if it is currently executing. If the command is not executing, nothing happens. Similarly, if an attempt to halt execution fails, nothing happens.

Usage

Use this method when you wish to cancel a command that is executing.

CreateParameter()

Returns

System.Data.IDbDataParameter

Parameters

None

Description

This method returns a valid parameter for the Command object.

Usage

Use this method whenever you wish to create a new parameter for a Command object.

Dispose()

Returns

Void

Parameters

None

Description

This method destroys the Command object.

Usage

Use this method when you no longer need the Command object and release the resources that it was occupying.

ExecuteNonQuery()

Returns

Integer: The number of rows affected

Parameters

None

Description

This method executes an SQL query through the Connection object property of the Command object.

Exception

InvalidOperationException: The connection to a data source does not exist, or it exists but is not open.

Usage

This is the ideal method to call whenever you issue a DDL command or update without the use of a DataSet. For such commands, the return value is -1.

When used with DML commands such as INSERT, UPDATE, and SELECT, the method returns the number of rows that were affected by the command.

ExecuteReader()

Returns

System.Data.IDataReader

Parameters

None

Description

This method executes the CommandText property of the Command object through the Connection object referred to by the Command's Connection property. Then it builds a valid IDataReader object with the resulting row set.

Usage

Use this method when you need to create and populate a DataReader object.

ExecuteScalar()

Returns

System.Object: A data value

Parameters

None

Description

This method executes the CommandText property of the Command object through its Connection object. Then it returns the value in the first column of the first row of the resulting row set.

Usage

Use this method when you need to obtain a single value from the database. A typical example would be to obtain a count of all records in the Customer table.

ExecuteXmlReader()**Returns**

System.Xml.XmlReader: A data value

Parameters

None

Description

This method executes the CommandText property of the Command object through its Connection object. Then it returns data as XML and populates a valid XmlReader object.

Usage

Use this method when you need to retrieve data as XML.

GetType()**Returns**

System.Type: A value indicating the type of the Command object

Parameters

None

Description

This method returns a valid type for the Command object.

ToString()

Returns

String: A valid representation of the Command object as a string

Parameters

None

The DataReader Object

The DataReader object is used to read data that has been retrieved from a database. While the Command and DataAdapter objects are used to issue database commands, the DataReader object is used to read retrieved data values. Once the values are inside this object, several other tasks can be achieved, such as:

- Populating a DataSet with retrieved data
- Populating a data binding control on a Windows Form or Web Form
- Assigning database values to client variables

The DataReader object should be the main interface between client objects and MTOs (Middle Tier Objects) because this object is equipped with some highly optimized functionality for data reading. The following section is a complete reference to the members of the DataReader object.

DataReader Properties

Depth

Type: Integer

Attribute: Read-only

Default: 0

Description: This property obtains a valid count of the number of nesting rows for the current row.

FieldCount

Type: Integer

Attribute: Read-only

Default: -1

Description: This property obtains the number of columns present in the current row that is being read.

IsClosed

Type: Boolean

Attribute: Read-only

Default: Null

Description: This property indicates whether the DataReader is closed or opened. It is used mostly to test the availability of data inside the object before issuing instructions to read the data.

Item(<column_name as string> or <column_ordinal as integer>)

Type: The native type of the item

Attribute: Read-only

Default: Null

Description: This property returns the value of the column specified by column_name or column_ordinal of the current row.

RecordsAffected

Type: Integer

Attribute: Read-only

Default: 0

Description: This property indicates how many records were affected by the execution of an SQL statement or changed on the client. The property returns 0 if there were no affected records or the SQL statement failed. -1 is returned for all SELECT statements.

DataReader Methods

Close()

Returns

Void

Parameters

None

Description

This method closes the DataReader object and frees up client resources.

CreateObjRef()

Returns

System.Runtime.Remoting.ObjRef: The returned object contains all the required information to generate a proxy that can communicate with another object residing remotely.

Parameters

System.Type: A valid type for the object to be created

Description

Call this method when you need to reference a valid object that contains all the required information to generate a proxy for the DataReader that will allow it to communicate with another object residing remotely. This method is used if you want the DataReader to communicate with remote objects.

Equals()**Returns**

Boolean

Parameters

obj: System.Object; any object inherited from the System.Object class

Description

Determines whether obj is equal to the DataReader object

Usage

This method will always return False if you pass an object that is not of the same type as the DataReader object. It is useful when you need to test whether two DataReader objects are holding exactly the same values from the same source.

GetBoolean()**Returns**

Boolean: The value of the specified column at the current row as a Boolean value

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

Description

If the column's type is not Boolean, the method does not attempt a conversion and throws an `InvalidCastException` exception. Be careful to test for a null value using the `IsDBNull()` method before attempting to read using this method.

GetByte()**Returns**

Byte: The value of the specified column at the current row as a byte value

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

Description

If the column's type is not byte, the method does not attempt a conversion and throws an `InvalidCastException` exception. Be careful to test for a null value using the `IsDBNull()` method before attempting to read using this method.

GetBytes()**Returns**

Integer: The count of Byte values read into the buffer parameter

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

dataIndex: Long; the index within the column from which the method will start reading Byte values into the buffer

buffer(): Byte; an array of type Byte that will hold the stream of byte values to be read

bufferIndex: Integer; the ordinal inside the buffer into which the first Byte value will be read

length: Integer; the maximum number of byte values to copy into the buffer

Description

This method is used to obtain a stream of byte values from a column in the DataReader.

GetChar()**Returns**

Char: The value of the specified column at the current row as a Char (single character) value.

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

Description

If the column's type is not Char, the method does not attempt a conversion and throws an InvalidCastException exception. Be careful to test for a null value using the IsDBNull() method before attempting to read using this method.

GetChars()

Returns

Integer: The count of Char values read into the *buffer* parameter

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

dataIndex: Long; the index within the column from which the method will start reading Byte values into the buffer

buffer(): Char; an array of type Byte that will hold the stream of Byte values to be read

bufferIndex: Integer; the ordinal inside the buffer into which the first Byte value will be read

length: Integer; the maximum number of Byte values to copy into the buffer

Description

This method is used to obtain a stream of Char values from a column in the DataReader.

GetDataTypeName()

Returns

String: The name of the type of column specified by column ordinal parameter *i*

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

Description

This method obtains the name of the type of column specified by the *i* parameter. It throws an `IndexOutOfRangeException` exception if the value specified by *i* was less than zero or greater than the `FieldCount` property of the `DataReader` object.

GetDateTime()**Returns**

System.DateTime: The value of the specified column at the current row as a `DateTime` object

Parameters

i: Integer; an index of the column. The `DataReader` contains a zero-based column array.

Description

If the column's type is not `DateTime`, the method does not attempt a conversion and throws an `InvalidCastException` exception. Be careful to test for a null value using the `IsDBNull()` method before attempting to read using this method.

If the *i* parameter is greater than the `FieldCount` property or less than zero, the method throws an `IndexOutOfRangeException` exception.

GetDecimal()**Returns**

System.Decimal: The value of the specified column at the current row as a `Decimal` object

Parameters

i: Integer; an index of the column. The `DataReader` contains a zero-based column array.

Description

If the column's type is not Decimal, the method does not attempt a conversion and throws an `InvalidCastException` exception. Be careful to test for a null value using the `IsDBNull()` method before attempting to read using this method.

If the `i` parameter is greater than the `FieldCount` property or less than zero, the method throws an `IndexOutOfRangeException` exception.

GetDouble()**Returns**

System.Double: The value of the specified column at the current row as a `System.Double` object

Parameters

i: Integer; an index of the column. The `DataReader` contains a zero-based column array.

Description

If the column's type is not Double, the method does not attempt a conversion and throws an `InvalidCastException` exception. Be careful to test for a null value using the `IsDBNull()` method before attempting to read using this method.

If the `i` parameter is greater than the `FieldCount` property or less than zero, the method throws an `IndexOutOfRangeException` exception.

GetFieldType()**Returns**

System.Type: The valid type of the specified column as a `System.Type` object

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

Description

This method is used to obtain a valid System.Type value that matches the data type of the field in the database.

GetFloat()**Returns**

System.Single: The value of the specified column at the current row as a System.Single object

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

Description

If the column's type is not Single, the method does not attempt a conversion and throws an InvalidCastException exception. Be careful to test for a null value using the IsDBNull() method before attempting to read using this method.

If the *i* parameter is greater than the FieldCount property or less than zero, the method throws an IndexOutOfRangeException exception.

GetGuid()**Returns**

System.Guid: The value of the specified column at the current row as a System.Guid object representing a globally unique identifier value

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

Description

If the column's type is not Guid, the method does not attempt a conversion and throws an `InvalidCastException` exception. Be careful to test for a null value using the `IsDBNull()` method before attempting to read using this method.

If the *i* parameter is greater than the `FieldCount` property or less than zero, the method throws an `IndexOutOfRangeException` exception.

GetHashCode()

Returns

Integer: The integer value indicating the hash code of the current row

Parameters

None

Description

Use this method to obtain the unique hash code for the DataReader.

GetInt16()

Returns

System.Int16: The value of the specified column at the current row as a `System.Int16` object (a 16-bit signed integer value)

Parameters

i: Integer; an index of the column. The `DataReader` contains a zero-based column array.

Description

If the column's type is not `Int16`, the method does not attempt a conversion and throws an `InvalidCastException` exception. Be careful to test for a null value using the `IsDBNull()` method before attempting to read using this method.

If the *i* parameter is greater than the `FieldCount` property or less than zero, the method throws an `IndexOutOfRangeException` exception.

GetInt32()**Returns**

System.Int32: The value of the specified column at the current row as a `System.Int32` object (a 32-bit signed integer value)

Parameters

i: Integer; an index of the column. The `DataReader` contains a zero-based column array.

Description

If the column's type is not `Int32`, the method does not attempt a conversion and throws an `InvalidCastException` exception. Be careful to test for a null value using the `IsDBNull()` method before attempting to read using this method.

If the *i* parameter is greater than the `FieldCount` property or less than zero, the method throws an `IndexOutOfRangeException` exception.

GetInt64()

Returns

System.Int64: The value of the specified column at the current row as a *System.Int64* object (a 64-bit signed integer value)

Parameters

i: Integer; an index of the column. The *DataReader* contains a zero-based column array.

Description

If the column's type is not *Int64*, the method does not attempt a conversion and throws an *InvalidCastException*. Be careful to test for a null value using the *IsDBNull()* method before attempting to read using this method.

If the *i* parameter is greater than the *FieldCount* property or less than zero, the method throws an *IndexOutOfRangeException*.

GetName()

Returns

String: The value of the name of the specified column at the current row as a *String* object

Parameters

i: Integer; an index of the column. The *DataReader* contains a zero-based column array.

Description

The *GetName()* method returns the name of the *DataSet* component and is useful when you want to refer to a *DataSet* by its name rather than using a reference variable.

GetOrdinal()

Returns

Integer: The ordinal of the column as an Integer value

Parameters

Name: String; the name of the column

Description

If the name specified is not a valid column name, the method throws an `IndexOutOfRangeException` exception.

GetSchemaTable()

Returns

System.Data.DataTable: A `DataTable` object containing the schema information for the columns of the `DataReader`

Parameters

None

Description

If the `DataReader` is closed, the method throws an `InvalidOperationException` exception.

GetString()

Returns

String: The value of the specified column at the current row as a `String` object

Parameters

i: Integer; an index of the column. The `DataReader` contains a zero-based column array.

Description

If the column's type is not `Int64`, the method does not attempt a conversion and throws an `InvalidCastException` exception. Be careful to test for a null value using the `IsDBNull()` method before attempting to read using this method.

If the `i` parameter is greater than the `FieldCount` property or less than zero, the method throws an `IndexOutOfRangeException` exception.

GetTimeSpan()**Returns**

System.TimeSpan: The value of the specified column at the current row as a `TimeSpan` object

Parameters

i: Integer; an index of the column. The `DataReader` contains a zero-based column array.

Description

If the column's type is not `TimeSpan`, the method does not attempt a conversion and throws an `InvalidCastException` exception. Be careful to test for a null value using the `IsDBNull()` method before attempting to read using this method.

If the `i` parameter is greater than the `FieldCount` property or less than zero, the method throws an `IndexOutOfRangeException` exception.

GetValue()**Returns**

System.Object: The value of the specified column at the current row as a `System.Object` object

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

Description

If the *i* parameter is greater than the FieldCount property or less than zero, the method throws an IndexOutOfRangeException exception.

GetValues()**Returns**

Integer: The count of Char values read into the buffer parameter

Parameters

values(): Object; an array of type Object that will hold the stream of Object values to be read

Description

This method is used to obtain a stream of Object values from a column in the DataReader. Be careful to test for a null value using the IsDBNull() method before attempting to read using this method.

IsDBNull()**Returns**

Boolean: A Boolean value indicating whether the value at the specified column in the current row is equal to a System.DBNull object. It's True if the value is null and False otherwise.

Parameters

i: Integer; an index of the column. The DataReader contains a zero-based column array.

Description

Use this method to check whether the value that you are about to read is null or not.

NextResult()**Returns**

Boolean: A Boolean value indicating whether there are any result sets from the current position.

Parameters

None

Description

Use this method when dealing with a batch of SQL statements that returned multiple sets of results. If this method returns True, there is a result set, and the method moves the DataReader object to its position.

Read()**Returns**

Boolean: A Boolean value indicating whether there are any rows in the DataReader. If this value is True, the DataReader has more rows and the position is shifted to the next row.

Parameters

None

Description

Use this method to test whether there are any rows inside the DataReader object, and move to the next row. The default location of the DataReader is prior to the first row, so you will have to call this method before you attempt any read operation on the DataReader.

- **Note:** .NET data providers such as the SQL Server .NET Data Provider provide several optimized methods that are equipped with functionality to read data of the types specific to that particular provider.

The DataAdapter Object

The DataAdapter object is used as a bridge between DataSet objects inside client applications and the data sources from which they were populated. The two key functions of the DataAdapter object are to update the data source with data inside the DataSet and vice versa.

This means that the DataAdapter is implemented to perform both retrieve and update operations on data sources. The generic `System.Data.Common.DbDataAdapter` class is quite limited in terms of the operations that it can perform. For this reason, we will look at a highly optimized DataAdapter class—the `System.Data.SqlClient.SqlDataAdapter` class.

- **Note:** The properties and methods that are not inherited from the generic DataAdapter class are marked appropriately.

SqlDataAdapter Constructor

The constructor is worth mentioning for the DataAdapter object. It overloads the generic constructor of the `DbDataAdapter` class and allows your application more flexibility to initialize a communication channel between client and database objects.

Taking advantage of the robust OOP concept of polymorphism, you can initialize a DataAdapter object in four different ways depending on the situation.

New()

Initializing a `DataAdapter` object this way simply creates a new instance of the `DataAdapter` object in memory. This is recommended when you are coding in a situation where you are not exactly sure what type of communication will take place between the data source and client ADO .NET objects.

Example:

```
Dim objDA as New System.Data.SqlClient.SqlDataAdapter()
```

New(System.Data.SqlClient.SqlCommand)

This type of initialization creates an instance of the `DataAdapter` object and initializes its `SelectCommand` property as a reference to the `SqlCommand` object passed as the parameter. Note that the `SelectCommand` property is not an initialized, new instance of `SqlCommand` in this case; rather it is a reference to the `SqlCommand` object that was passed as a parameter. This means that the two objects are pointing to the same value in memory.

The database command referenced by the `SelectCommand` property is directly executed against the database, and the result set is stored inside the `DataAdapter` object.

This type of initialization is used when you already know what type of command needs to be executed against the database and the command is already prepared.

Example:

```
Dim objConn as SqlConnection = New SqlConnection  
    ("DataSource=LOCALHOST;Database=Northwind;  
    UserID=sa;password=sa")  
  
Dim objCmd as SqlCommand = New SqlCommand("SELECT * FROM  
    Employees")  
  
Dim objDA as SqlDataAdapter = New SqlDataAdapter(objCmd)
```

New(command as String, connection as String)

This type of initialization creates an instance of the `DataAdapter` object by specifying a command text and connection text that it uses to connect and issue commands to a database.

The `DataAdapter` object first connects to the database using the parameters contained within the connection string parameter. To perform that action, it utilizes an internally optimized `SqlConnection` object.

The command parameter is used to issue an SQL command to the database through the object's `SelectCommand` property. The SQL command must be an SQL `SELECT` command or a command that executes a stored procedure expected to return results.

Example:

```
Dim strCommand as String
Dim strConn as String

strConn = "DataSource=LOCALHOST;Database=Northwind;
          UserID=sa;password=sa"
strCommand = "SELECT * FROM Employees"

Dim objDA as SqlDataAdapter = New SqlDataAdapter
(strCommand, strConn)
```

New(command as String, connection as SqlConnection)

This type of initialization creates an instance of the `DataAdapter` object by specifying a command text and a valid connection object that it uses to connect and issue commands to a database.

The `DataAdapter` object uses the connection parameter to connect to the database. Note that the connection

parameter must already contain an open connection to the database.

The command parameter is used to issue an SQL command to the database through the object's `SelectCommand` property. The SQL command must be an SQL `SELECT` command or a command that executes a stored procedure expected to return results.

Example:

```
Dim strCommand as String
Dim objConn as SqlConnection = New SqlConnection
    ("DataSource=LOCALHOST;Database=Northwind;
    UserID=sa;password=sa")

strCommand = "SELECT * FROM Employees"

Dim objDA as SqlDataAdapter = New
    SqlDataAdapter(strCommand, objConn)
```

SqlDataAdapter Properties

AcceptChangesDuringFill

Type: Boolean

Attribute: Read/Write

Default: Null

Description: This property is used to determine whether the `DataRow` method `AcceptChanges()` is called when a new `DataRow` is added to a `DataTable` within the `DataAdapter` object.

If this property is set to `True`, the `AcceptChanges()` method is called immediately after every `DataRow` object is added to the `DataTable` object.

ContinueUpdateOnError

Type: Boolean

Attribute: Read/Write

Default: Null

Description: This property is used to determine whether the DataAdapter should generate an exception when an error occurs while updating a row in the database or continue with the update of the same row and any other rows.

When this property is set to True, the DataAdapter object continues updating rows and does not generate any exception or error messages when an error occurs.

DeleteCommand

Type: SqlCommand

Attribute: Read/Write

Default: Null

Description: This property is a reference to an SqlCommand object that is optimized to delete data from a database. The CommandText property of that object must be either an SQL statement that deletes data from a database or a call to a stored procedure that performs this function. This is important because it is impossible to return values using this property. A typical scenario would be to prepare the SqlCommand object in code and then assign it to this property.

InsertCommand

Type: SqlCommand

Attribute: Read/Write

Default: Null

Description: This property is a reference to an SqlCommand object that is optimized to insert data into a database. The CommandText property of that object must be either an SQL statement that inserts data from a database or a call to a stored procedure that performs this function. This is important because it is impossible to return values or delete data using this property. A typical scenario would be to prepare the SqlCommand object in code and assign it to this property.

MissingMappingAction

Type: MissingMappingAction enumeration

Attribute: Read/Write

Default: Null

Description: This property is used to evaluate which action the DataAdapter performs when data that it is given does not have a valid mapping or match a DataTable object inside a DataSet object. The possible values are:

Error: An exception is generated by the DataAdapter.

Ignore: The DataAdapter ignores the error and inserts a null value into the column.

Passthrough: The DataAdapter causes the DataSet to create a column or table matching the unmatched value.

MissingSchemaAction

Type: MissingSchemaAction Enumeration

Attribute: Read/Write

Default: Null

Description: This property is used to evaluate which action the DataAdapter performs when data that it is given does not have a valid mapping to any existing DataTable or DataColumn object inside a DataSet. In other words, the DataTable or DataColumn that is referred to is missing. The possible values are:

Add: The DataAdapter adds the DataColumn or DataTable object to the DataSet.

AddWithKey: The DataAdapter recreates the schema for the DataSet so that it incorporates information for the missing DataTable or DataColumn object.

Ignore: The DataAdapter ignores the missing DataColumn or DataTable object(s) that the code is referring to. In this scenario, data will be lost.

Error: Generates an exception

SelectCommand

Type: SqlCommand

Attribute: Read/Write

Default: Null

Description: This property is a reference to an SqlCommand object that is optimized to retrieve data from a database. The CommandText property of that object must be either an SQL statement that retrieves data from a database or a call to a stored procedure that performs this function. This is important because it is impossible to delete or insert data using this property. A

typical scenario would be to prepare the `SqlCommand` object in code and assign it to this property.

TableMappings

Type: `DataTableMappingCollection`

Attribute: Read-only

Default: Null

Description: This property is a collection of all the available table mapping information that is present between `DataTable` objects within a `DataSet` and the `Table` objects in the database from which data has just been retrieved by the `DataAdapter`. This data is about to be filled into the `DataSet`.

UpdateCommand

Type: `SqlCommand`

Attribute: Read/Write

Default: Null

Description: This property is a reference to an `SqlCommand` object that is optimized to update data from a `DataSet` into a database. The `CommandText` property of that object must be either an SQL statement that updates data into a database or a call to a stored procedure that performs this function. This is important because it is impossible to retrieve, delete, or insert data using this property. A typical scenario would be to prepare the `SqlCommand` object in code and assign it to this property.

SqlDataAdapter Methods

CreateObjRef()

Returns

System.Runtime.Remoting.ObjRef: The returned object contains all the required information to generate a proxy that can communicate with another object residing remotely.

Parameters

System.Type: A valid type for the object to be created

Description

Call this method when you need a reference to a valid object that contains all the required information to generate a proxy for the DataAdapter, which will allow it to communicate with another object residing remotely. This method is used if you want the DataAdapter to communicate with remote objects.

Dispose()

Returns

Void

Parameters

None

Description

This method releases all the processing resources that the DataAdapter object holds.

Fill()

Returns

Integer: The number of rows that have been added or updated inside the DataSet

Parameters

This method has a very solid polymorphic mechanism. The actions performed by this method hinge on the type of parameters passed to it. The following gives a list of parameters and their corresponding descriptions.

System.Data.DataTable: A DataTable object into which data is to be copied.

Use this parameter only when you need to fill a single DataTable object for use on the client.

System.Data.DataSet: A DataSet object into which data is to be copied.

Use this parameter when you need to fill an entire DataSet object for use on the client. A typical scenario is when the DataAdapter contains multiple result sets.

System.Data.DataSet: A DataSet object into which data is to be copied.

String: A string containing the name of a source table that contains the mapping information for the retrieved result set.

Use this parameter when you need to fill an entire DataSet object for use on the client. The schema of the database will also be created inside the DataSet using the information from the source table. A typical scenario is when the DataAdapter contains multiple result sets and you also want to put a schema inside the DataSet.

System.Data.DataSet: A DataSet object into which data is to be copied

Integer: The record at which you want to start populating

Integer: The maximum number of records to populate

String: A string containing the name of a source table that contains the mapping information for the retrieved result set

FillSchema()

Returns

System.Data.DataTable: A valid DataTable object into which the DataAdapter populates a valid schema

Parameters

This method also has a very solid polymorphic mechanism. The actions performed by this method hinge on the type of parameters passed to it. The following gives a list of parameters and their corresponding descriptions.

System.Data.DataTable: A DataTable object into which the schema is copied

SchemaType: A valid value from the SchemaType enumeration.

Use this parameter only when you need to fill a single DataTable object to contain the schema information of a result set.

System.Data.DataSet: A DataSet object into which the schema is copied

SchemaType: A valid value from the SchemaType enumeration.

Use this parameter only when you need to fill a single DataSet object to contain the schema information of a result set.

System.Data.DataSet: A DataSet object into which to copy the schema

SchemaType: A valid value from the SchemaType enumeration

String: A string value containing the name of a source table that contains the mapping information for the retrieved result set.

Use this parameter only when you need to fill a single DataSet object to contain the schema information of a result set based on table mapping information.

Update()

Returns

Integer: An integer value indicating the number of rows that were affected by the Update() method

Parameters

This method also has a very solid polymorphic mechanism. The actions performed by this method hinge on the type of parameters passed to it. The following gives a list of parameters and their corresponding descriptions.

System.Data.DataSet: The DataSet that you wish to use to update a data source.

Passing this parameter instructs the method to update the data source with the data values found inside the DataSet object.

System.Data.DataRow: An array of DataRow objects that you wish to use to update a data source.

Passing this parameter instructs the method to update the data source with the data values found inside an array of DataRow objects.

System.Data.DataTable: The `DataTable` that you wish to use to update a data source.

Passing this parameter instructs the method to update the data source with the data values found inside the `DataTable` object.

System.Data.DataSet: The `DataSet` that you wish to use to update a data source.

String: The name of the source table that is used for table/`DataColumn` mapping.

Passing this parameter instructs the method to update the data source with the data values found inside the `DataSet` object. If table mapping is necessary, the table specified by the second parameter is used for that purpose.

Summary

This chapter is your complete reference to the database—to client objects that are part of the ADO .NET class library. As you can clearly see, ADO .NET provides you with a very powerful set of objects used to communicate with databases. If you want a completely practical tutorial of how to interact with databases in ADO .NET, read Chapter 7.

The next chapter is also a theoretical reference to the objects within ADO .NET that allow you, the developer, to take control of data manipulation in your client application.

Data Manipulation

In This Chapter

In the previous chapter, you read about the ADO .NET components that allow .NET applications to interact with data sources. A complete reference was provided for the Connection, Command, DataReader, and DataAdapter components. These components are the primary interface to data sources in .NET programming. This chapter is about the second group of ADO .NET components that enables developers to manipulate data inside client applications:

- DataSet
- DataTable
- DataRow
- DataView
- DataColumn

A complete reference is available for the DataSet component.

Although these components are implemented as independent and stand-alone classes, they are all tightly integrated into the DataSet, which is the most instrumental part of the disconnected data access architecture of ADO .NET. For the sake of clarity and organization, the reference sections of this chapter treat each component independently. While a complete reference is provided on the DataSet component, the other components are briefly

covered by the end of the chapter. Chapter 8 provides a more practical view of how all these components are organized to tell one story inside the DataSet component. It is important to keep in mind that the DataSet is an optimized and organized way to make the other components work together. For you to better understand the architectural design, the chapter focuses on the structural design of the DataSet component.

The DataSet Component

What is the DataSet?

It seems safe to think about the DataSet component as an in-memory form of a database residing inside a client application instead of being hosted by a database server. In fact, sticking to the strict design goals of the component, this is true. The data manipulation components listed above are all client representations of the database objects one would come across during the course of database design. The database becomes a proper analogy for the DataSet because, just as the former organizes its objects, the DataSet similarly provides a seamlessly integrated and structured code environment for all the client data components to interact concertedly. By exposing the data in a way similar to how it would appear inside the database, the DataSet component brings the true power of database programming to the web-distributed development model.

When Do You Need the DataSet?

You do not need to utilize a DataSet component for every client application. Consider the case when you simply need a read-only and forward-only list of items to place in a ComboBox control on a Web Form object. Creating a DataSet to perform this kind of task is a waste of client resources and may have a huge impact on application performance. You may be thinking about populating a DataTable object for such as task. But even then, resources are wasted. In this case, it is better to use a DataReader component—covered in Chapter 2—and loop through its set of retrieved rows to place items into the control.

In general, you should use a DataSet when you need to perform the following tasks:

- Retrieve data that you plan to save back to the database. The DataSet and DataAdapter components have a seamless interoperability architecture that allows the DataSet to interact with the database without having any knowledge of the data source (hence the notion of Disconnected Data Source that is central to the .NET Framework's support for distributed application development).
- Convert data from a relational form to a hierarchical form, such as XML. ADO .NET provides tight integration with XML—covered in Chapter 6, “Practical ADO .NET Programming (Part One).”
- Combine data from different data sources. The DataSet component allows you to combine data from a limitless number of sources.
- Post data to remote components across the Internet. It is possible for remote components to communicate via DataSet components or web services created from DataSet components.

How is the DataSet Organized?

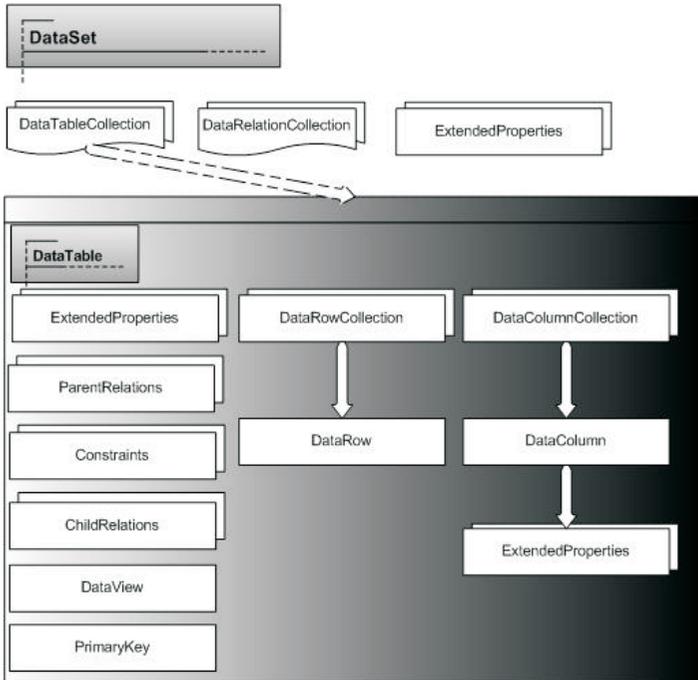


Figure 3-1: The organization of the DataSet component

Throughout the rest of this chapter, the component that is central to our discussion is the DataSet component. All the components shown in Figure 3-1 are implemented independently. However, it is important to obtain a solid understanding of the beauty of the DataSet; therefore, all the components are described as they appear in the object model of the DataSet. You will notice that there is not much code in this chapter. This is a reference chapter with a theoretical approach to explaining client data manipulation. Since practical exposure is also very important, Chapter 6 had been dedicated entirely to providing practical experience. Chapter 6 provides information about how to:

- Populate the DataSet
- Read DataSet content
- Save DataSet content

Before the subtle components within the DataSet are covered, let's examine the properties and methods of the DataSet itself.

Core DataSet Properties

CaseSensitive

Type: Boolean

Attribute: Read/Write

Default: ""

Description: The CaseSensitive property of the DataSet object defines whether any string comparison performed on any data value(s) within the DataSet would be case sensitive or otherwise.

Possible values:

TRUE: String comparisons performed on the DataSet values are case sensitive.

FALSE: String comparisons performed on the DataSet values are not case sensitive.

Container

Type: System.ComponentModel.Container

Attribute: Read-only

Default: ""

Description: The Container property of the DataSet object obtains a valid reference to the object that contains the DataSet.

DataSetName

Type: String

Attribute: Read/Write

Default: ""

Description: The DataSetName property of the DataSet object is used to specify a name for the DataSet object. Setting or getting a name for a DataSet is important if you wish to refer to the particular DataSet by name in the code.

DefaultViewManager

Type: System.Data.DataViewManager

Attribute: Read-only

Default: ""

Description: The DefaultViewManager property of the DataSet object is used to obtain a reference to the default view and organization of data within the DataSet. A DataViewManager object maintains a collection of views for all the DataTable objects within a DataSet.

DesignMode

Type: Boolean

Attribute: Read-only

Default: ""

Description: The DesignMode property of the DataSet object is used to determine whether the particular DataSet is in design mode.

EnforceConstraints

Type: Boolean

Attribute: Read/Write

Default: ""

Description: The EnforceConstraints property is used to indicate whether the DataSet should enforce its defined constraints on data values that it receives when it is being updated. As an in-memory data source, the DataSet component can have constraints defined similar to a data source on a database server. Each DataTable component that is part of a particular DataSet can have its constraints defined using the DataTable's Constraints property.

ExtendedProperties

Type: System.Data.PropertyCollection

Attribute: Read/Write

Default: ""

Description: The ExtendedProperties property is used to return a collection of custom properties that has been defined for the particular DataSet component. It is also possible to add a custom property to the DataSet using this property.

HasErrors

Type: Boolean

Attribute: Read-only

Default: False

Description: The HasErrors property is used to indicate whether there are any error(s) in any row of a DataTable component inside the particular DataSet component.

Locale

Type: System.Globalization.CultureInfo

Attribute: Read

Default: Null

Description: The Locale property is used to obtain information about the locale of the user's machine. This is a very useful property, as it allows you to define custom actions based on the user's locale preferences.

Namespace

Type: String

Attribute: Read/Write

Default: ""

Description: The Namespace property is used to indicate the name of the namespace to which the particular DataSet belongs. This property is most useful when you want to use XML functions to populate an XML document from one or more DataSet components.

Prefix

Type: String

Attribute: Read/Write

Default: ""

Description: The Prefix property is used to indicate an XML prefix of the DataSet component's namespace when an XML document is created. The Namespace property should be set prior to defining the Prefix property.

Relations

Type: System.Data.DataRelationCollection

Attribute: Read-only

Default: Empty Collection

Description: The Relations property is a collection of DataRelation components that are used within the DataSet component to define the relations between the DataSet's DataTable components.

Site

Type: System.ComponentModel.ISite

Attribute: Read/Write

Default: Null

Description: The Site property is used to indicate a valid ISite reference for the DataSet. The ISite class enables seamless communication and integration between a .NET component and its container, such as a parent class, Windows Form, or Web Form component.

Tables

Type: System.Data.DataTableCollection

Attribute: Read-only

Default: Null

Description: The Tables property is used to obtain a collection of all DataTable component(s) that are available within the DataSet component. If no DataTable component is available within the DataSet, the property has a value of NULL.

As you saw in Figure 3-1, the `DataTableCollection` component within the `DataSet` has an entire object model of its own that allows seamless manipulation of data within the `DataTable` components of a `DataSet`. This collection and its content are fully covered later in the chapter.

Core DataSet Methods

AcceptChanges()

Returns

None

Parameters

None

Description

This method instructs the `DataSet` component to commit all the changes that have been made to its data values since the `DataSet` was last populated or the last call to this same method was issued.

Usage

Call this method to commit all the changes that have been made to a `DataSet` component. This is useful before you use the `DataSet` to perform any data transmission to an XML document, a remote object, or the data source through the `DataAdapter`.

BeginInit()

Returns

None

Parameters

None

Description

This method initializes a DataSet component.

Usage

Call this method to initialize a DataSet component when it is contained by another component, a Web Form, or a Windows Form object. Automatic initialization does not occur in any of the three components.

Clear()**Returns**

None

Parameters

None

Description

This method removes all the rows of all DataTable components within the DataSet component. Thus, it empties the DataSet component of all its data values.

Usage

Call this method when an ultimate housecleaning is needed for a DataSet component. Note that a call to this method does not mean that all DataTables or any of their relations are deleted from the DataSet. These would remain intact until you alter them or destroy the DataSet component.

Clone()

Returns

System.Data.DataSet: A component that is a direct clone of the particular DataSet

Parameters

None

Description

This method is used to create a new DataSet component that is similar to the particular DataSet on which the method is called.

Usage

Use this method when you need to clone the current DataSet component. Note that when you clone the DataSet, no data values are copied across into the new DataSet component created. Only the structure of the DataSet is copied into the new DataSet. If you need to copy data and structure, call the Copy() method.

Copy()

Returns

System.Data.DataSet: A component that is a direct clone and has all the data values of the particular DataSet

Parameters

None

Description

This method is used to create a new DataSet component that is similar to and contains the same data values as the particular DataSet on which the method is called.

Usage

Use this method when you need to clone the current DataSet component being used and copy all of its data values into the clone. If you need to create a clone without the data values, call the Clone() method.

Dispose()**Returns**

None

Parameters

None

Description

This method is used to destroy the DataSet component and release the system processing resources that it holds.

Usage

This method is useful to effectively manage client resources that a DataSet component is currently utilizing.

EndInit()**Returns**

None

Parameters

None

Description

This method stops the initialization of a DataSet component.

Usage

Call this method to stop the initialization of a DataSet component when it is contained by another component, a Web Form, or Windows Form object.

GetChanges()**Returns**

System.Data.DataSet: A DataSet component that contains all the changes that had been made since the last call was made to the AcceptChanges() method or the DataSet was loaded

Parameters

None

Description

This method is used to create a new DataSet component that holds all the changes that had been made to the current DataSet component.

Usage

Use this method when you need to create a DataSet component that contains the changes made by a user to the current DataSet component. This is a good way to allow the application to support undo functionality on the DataSet.

GetService()**Returns**

System.Object: An object or component that is a service

Parameters

System.Object: An object or component that implements the *System.IServiceProvider* interface

Description

This method is used to obtain a reference to another component or object that is provided as a service by either the *DataSet* component or the component or object that is the container of the *DataSet*.

Usage

Use this method when you need to reference a service in your code that would perform some form of desired task or provide data.

GetType()**Returns**

System.Type: A valid data type

Parameters

None

Description

This method is used to obtain knowledge of the data type of any component or object in the .NET Framework. In the case of the *DataSet*, it would return *System.DataSet*.

Usage

The *GetType()* method is a generic method inherited from the *System.Object* class. It returns a valid *System.Type* value that is the data type of the object on which it is called.

GetXML()

Returns

String: A string of the XML representation of the DataSet's data

Parameters

None

Description

This method is used to obtain a string of characters formatted as XML that is the representation of all the data that the DataSet component contains.

Usage

Use this method when you wish to obtain a DataSet's data as a string of XML tags. Note that by using this method, you are simply obtaining an XML string. To write data to a file, you would have to use the WriteXML() method. To obtain a full comprehensive overview of XML integration with ADO .NET, read Chapter 5, "XML Integration with ADO .NET."

GetXmlSchema()

Returns

String: A string of the XSD schema for the XML that would represent the DataSet's data

Parameters

None

Description

This method is used to obtain a string of characters formatted as XSD schema that is the schema of the XML

that is the representation of all the data that the DataSet component contains.

Usage

Use this method when you wish to obtain a DataSet's XML schema as a string. Note that by using this method, you are simply obtaining a string. To write the schema to a file, you would have to use the WriteXMLSchema() method. To obtain a full comprehensive overview of XML integration with ADO .NET, read Chapter 5.

HasChanges()

Returns

Boolean: A value that indicates whether there have been any changes made to the DataSet component

Parameters

The default method does not take any parameters. There is an overload to this method that takes the following parameter(s):

System.Data.DataRowState: This enumeration defines the type of changes that you are looking for. Since the HasChanges() method acts on DataRow components of the DataSet's DataTable components, the DataRowState enumeration defines the state of a DataRow component. Here is a list of possible values of DataRowState and the resulting effect on the HasChanges() method.

Added: The HasChanges() method returns True if there has been any DataRow component(s) added to any DataTable component(s) of the DataSet.

Deleted: The HasChanges() method returns True if there has been any DataRow component(s) deleted from any DataTable component(s) of the DataSet.

Detached: The HasChanges() method returns True if there has been any DataRow component(s) detached from any DataTable component(s) of the DataSet.

Modified: The HasChanges() method returns True if there has been any DataRow component(s) amended in any DataTable component(s) of the DataSet.

Unchanged: This is a bit confusing. You expect the HasChanges() method to return True only if changes have been made to the DataSet. If Unchanged is passed as a filter, the method will return True if the DataSet has not changed and False if it has! This might have some undesirable effect on your application.

Description

This method returns a value indicating whether the data inside the DataSet component has changed from the last time that the AcceptChanges() method was called on the DataSet.

Usage

Use this method to check whether the DataSet contains new data. It is usually very useful before an update occurs.

InferXmlSchema()

Returns

None

Parameters

The parameter list that is passed totally depends on the overloaded method that you wish to call:

Overload 1:

System.IO.Stream: This is a Stream component or any other component inheriting from it. A Stream component provides the most generic services for conducting input/output operations with a sequence of bytes.

The DataSet would read its schema information from this Stream component.

String(): An array of String values that contain the namespace URIs that are to be excluded from the schema inference process

Overload 2:

String: A String value that represents the fully qualified path of a file from which the DataSet would infer its schema

String(): An array of String values that contain the namespace URIs that are to be excluded from the schema inference process

Overload 3:

System.IO.TextReader: This is a TextReader component or any other component inheriting from it. A TextReader component provides the most generic services for reading a sequence of characters.

The DataSet would read its schema information from this TextReader component.

String(): An array of String values that contain the namespace URIs that are to be excluded from the schema inference process

Overload 4:

System.Xml.XmlReader: An XmlReader component or any other component inheriting from it. An

XmlReader component provides forward-only access to XML data.

The DataSet would read its schema information from this XmlReader component.

To obtain a full comprehensive overview of XML integration with ADO .NET, read Chapter 5, “XML Integration with ADO .NET.”

String(): An array of String values that contains the namespace URIs that are to be excluded from the schema inference process

Description

This method is used to infer the XML schema from a TextReader or XmlReader component or a file into the DataSet.

Usage

Use this method when you need to infer the XML schema of a TextReader component or a file into the DataSet.

Merge()

Returns

None

Parameters

The parameter list that is passed totally depends on the overloaded method that you wish to call:

Overload 1:

System.Data.DataRow(): This is an array of DataRow components or any other component inheriting from it.

The DataSet would merge its data with the data provided by this array. Merging in this case would be that the DataSet imports data from the DataRow component(s).

Overload 2:

System.Data.DataSet: This is a DataSet component or any other component inheriting from it.

The DataSet on which this method is called would merge its data and schema with those provided by this DataSet.

This is a very powerful way to refresh data inside a client application.

Overload 3:

System.IO.TextReader: This is a TextReader component or any other component inheriting from it. A TextReader component provides the most generic services for reading a sequence of characters.

The DataSet would read its schema information from this TextReader component.

String(): An array of String values that contains the namespace URIs that are to be excluded from the schema inference process

Overload 4:

System.Xml.XmlReader: An XmlReader component or any other component inheriting from it. An XmlReader component provides forward-only access to XML data.

The DataSet would read its schema information from this XmlReader component.

To obtain a full comprehensive overview of XML integration with ADO .NET, read Chapter 5, “XML Integration with ADO .NET.”

String(): An array of String values that contains the namespace URIs that are to be excluded from the schema inference process

Description

Calling the Merge() method causes the DataSet to import the data from the objects that are passed as parameters and merge it with its own data. This is a good way to merge data from two separate DataSet components that have the same schema. Most merge operations are done before the database is updated.

ReadXml()

Returns

XmlReadMode: The valid XML read mode used to read the data

Parameters

Stream: A Stream object or any object that derives from the Stream class. This parameter is more often a reference to an XML document.

Description

This method is used to obtain a string of characters formatted as XML that is to be stored inside the DataSet component.

Usage

Use this method when you wish to populate a DataSet with an XML document or any form of XML source. To write data to a file, you would have to use the WriteXML() method. To obtain a full comprehensive overview of XML integration with ADO .NET, read Chapter 5.

ReadXmlSchema()

Returns

None

Parameters

Stream: A Stream object or any object that derives from the Stream class. This parameter is more often a reference to an XML document.

Description

This method is used to obtain a string of characters formatted as the schema of an XML document that is to become the schema of the data stored inside the DataSet component.

Usage

Use this method when you wish to apply a particular XML schema to a DataSet from the schema of an XML document or any form of XML source. To obtain a full comprehensive overview of XML integration with ADO .NET, read Chapter 5.

RejectChanges()

Returns

None

Parameters

None

Description

This method is used to reject or roll back all the changes that have occurred to the DataSet component since its AcceptChanges() method was last called.

Usage

Use this method when you wish to reject all the changes that occurred to a DataSet. This is a powerful functionality that allows for undo operations and rollback facilities on the client machine.

Reset()**Returns**

None

Parameters

None

Description

This method is used to reset the DataSet to its original state.

Usage

Use this method when you wish to perform a full rollback of changes made to the DataSet from the time it was initialized in memory.

ToString()**Returns**

None

Parameters

None

Description

This method is used to obtain the representation of the DataSet as a string.

Usage

Use this method when you wish to obtain the representation of the DataSet as a string. The string is not in any specific format. If you want the DataSet's data as an XML stream, use the GetXml() method.

WriteXml()**Returns**

None

Parameters

Stream: A Stream object or any object that derives from the Stream class. This parameter is more often a reference to an XML document.

Description

This method is used to write the content of a DataSet component to a file as XML.

Usage

Use this method when you wish to obtain XML inside a file from a DataSet's content. It is important to initialize the Stream parameter first. To obtain a full comprehensive overview of XML integration with ADO .NET, read Chapter 5, "XML Integration with ADO .NET."

WriteXmlSchema()**Returns**

None

Parameters

Stream: A Stream object or any object that derives from the Stream class. This parameter is more often a reference to an XML document.

Description

This method is used to write the structure of a DataSet component as XML schema to a file.

Usage

Use this method when you wish to obtain XML schema inside a file from a DataSet's structure. It is important to initialize the Stream parameter first. To obtain a full comprehensive overview of XML integration with ADO .NET, read Chapter 5, "XML Integration with ADO .NET."

DataSet.ExtendedProperties

This section of the chapter takes an in-depth look at the ExtendedProperties property of the DataSet component. Since this property is of type System.Data.PropertyCollection, most of what you read here will apply to any component or object inheriting from the System.Data.PropertyCollection class.

The ExtendedProperties property is worth discussing in this chapter because it is equipped with properties and methods of its own that provide you with many ways to customize, extend, and, therefore, add power to the generic DataSet component. To use ExtendedProperties effectively, you need to know two basic things:

- How to add an extended property to a DataSet
- How to read and write values to the property

Adding an Extended Property

To add an extended property to a DataSet, use the Add() method. The following example adds two extended properties to a DataSet:

```
Dim ds as New DataSet ()

'Initialize DataSet
With ds.ExtendedProperties

    .Add ("username", "Terrence")
    .Add ("password", "pizza")

End With
```

Reading and Writing Values

There are several ways to read the values of the extended properties.

To read the value of one single extended property, use the Item(*index*) property:

```
With ds.ExtendedProperties

    MsgBox(.Item(0)) 'Display message box showing the value of
                    'the username property – "Terrence"

End With
```

To obtain the values of all the extended properties as a collection, use the Values property to populate an ICollection component:

```
Dim propColl as ICollection

With ds.ExtendedProperties
    propColl = .Values
End With
```

DataTableCollection

As mentioned earlier, a DataSet is a container of data. Its data are organized inside objects of type DataTable. The DataTableCollection property of the DataSet is simply an optimized ICollection component that acts as a container of DataTable objects within the DataSet.

The advantages of maintaining a collection of DataTable objects are as follows:

- It allows the DataSet to provide a convenient way for consumers to refer to all its content as a single object.
- It allows consumers to sort the content of a DataSet based on a specific member of the DataTable components contained within the collection.
- It allows consumers to easily iterate through the content of a DataSet and perform actions.

This component is very complex and easily manageable. A developer does not have to use this component property directly because the operations of adding and deleting DataTable objects are done entirely by the DataSet component.

DataRelationCollection

A DataSet acts as in-memory data storage and a container of DataTable components. For the proper management of relationships between items of related DataTable components that it contains, the DataSet maintains a collection of DataRelation components inside another component called DataRelationCollection. This component is another optimized version of the generic ICollection component.

A relationship between a DataColumn of one DataTable to that of another DataTable allows for the association of DataRow objects between the two tables. A typical relationship would be Order and OrderItems.

The advantages of using a DataRelationCollection are:

- You are better able to navigate from one table to another within the DataSet simply by referencing the DataRelation's key within the collection.
- It allows for integrity enforcement by providing you with ways to specify UniqueKeyConstraint and ForeignKeyConstraint component properties for each DataRelation. Of course, this can be done on a specific DataRelation component, but it is more manageable when everything can be done from the same location.

The Big Picture

Much has been said about the DataTable component throughout this book. In this section, this component is not dissected in detail as previously done with the DataSet, but you are given the tools you need to work with the DataTable.

The DataTable is the real container of data items in ADO .NET's client components. Think of it as a database table. Data are stored in components called DataRow, which are analogous to a table's row in database terms. The DataRow itself can have one or several DataColumnns.

ADO .NET goes further by providing for the instrumentation of different views of the data stored by a DataTable through components called DataViews. The following example shows you how to create a DataTable component, assign columns to it, populate it, and create a view for it. The view is then used as a data source for a DataGrid on a Windows Form object. To view and perhaps extend this code, see the Chapter03 folder on the companion CD. The example uses the Northwind database in SQL Server 2000.

```
Public Class frmCustomers
    Inherits System.Windows.Forms.Form

    Private dtCustomers As New DataTable()
    Private dvCustomers As DataView

    #Region " Windows Form Designer generated code "

    Private Sub cmdPopulateGrid_Click _
        (ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles _
        cmdPopulateGrid.Click

        PopulateGrid() 'Populate the grid

    End Sub

    Private Function PopulateGrid()

        If dtCustomers.Rows.Count < 1 Then
            PrepareData() 'Prepare the datasource
            'Assign the data source to the grid
            With dgCustomers
                .DataSource = dvCustomers
            End With
        Else
            'If there are rows in the DataTable, there is
            'no need to retrieve again
            lblMsg.Text = "The grid is already populated."
        End If

    End Function

    Private Function PrepareData()

        Dim dr As DataRow
        Dim objCustReader As Data.SqlClient.SqlDataReader

        'Create columns in the DataTable
        With dtCustomers.Columns
            .Add(New DataColumn("Customer ID", _
                GetType(String)))
            .Add(New DataColumn("Company Name", _
                GetType(String)))
        End With
    End Function
End Class
```

```

        .Add(New DataColumn("Contact Name", _
                            GetType(String)))
        .Add(New DataColumn("Address", _
                            GetType(String)))
    End With

    'Populate the DataReader
    objCustReader = GetData()

    'Populate the DataTable
    With objCustReader
        While .Read()
            'A new DataRow is added upon each
            'iteration
            dr = dtCustomers.NewRow()
            dr(0) = .Item("CustomerID")
            dr(1) = .Item("CompanyName")
            dr(2) = .Item("ContactName")
            dr(3) = .Item("Address")
            dtCustomers.Rows.Add(dr)
        End While
    End With

    'Create a view of the DataTable
    dvCustomers = New DataView(dtCustomers)

    'Free Resources
    objCustReader = Nothing

End Function

Private Function GetData() As _
    Data.SqlClient.SqlDataReader

    Dim strConn As String
    Dim strQuery As String
    Dim objConn As New Data.SqlClient.SqlConnection()
    Dim objCmd As New Data.SqlClient.SqlCommand()
    Dim objReader As Data.SqlClient.SqlDataReader

    'Set up the connection string for the
    'SQL Server database
    strConn = "Data Source=LOCALHOST;"
    strConn &= "Initial Catalog=Northwind;"
    strConn &= "User ID=sa;password=;"

```

```
With objConn
    'Assign connection string
    .ConnectionString = strConn
    'Open a connection to the database
    .Open()
End With

'Set up the query string
strQuery = "SELECT CustomerID, CompanyName, "
strQuery &= " ContactName, Address "
strQuery &= " FROM Customers"

'Execute the command
With objCmd
    .CommandText = strQuery
    .Connection = objConn
    objReader = .ExecuteReader()
    .Dispose()
End With

'Return the reader
Return objReader

End Function

End Class
```

Summary

The DataSet is the key element of data manipulation inside client applications. It is important to keep in mind that when performing simple tasks, such as populating a grid like in the previous example, you need to consume as few client resources as possible. Avoid using a complex component like the DataSet in this case and simply go for a DataTable. There are instances, of course, when you would want to bind the grid to a DataSet because you would want the user to edit the data and save it back to the data source. You are now advised to read Chapter 8, “Migrating ADO Applications,” which provides you several scenarios of common data manipulation tasks.

Designing ADO .NET Applications

.NET Application Models

The .NET Framework supports a variety of application architectures. In this chapter, we examine where and how to use ADO .NET in the different application architectures of the .NET Framework. First, though, let's have a look at the different architectures that you can use in .NET.

There are mainly four kinds of applications that you can build:

- Windows Forms applications
- Console applications
- Windows Services applications
- ASP .NET web applications

Windows Forms Applications

This is the classic Windows application. The user interfaces are done through Windows Forms and Windows Form Controls, which are fully object oriented. In nearly all of the cases, these types of applications involve some sort of data modification in the application, and the modified data is then stored in a data source. The data source can be a database or a file, as is the case with a word processor.

Database client-server applications are also part of this architecture. The application is usually installed on the client's machine, and connections are made directly to the databases using ADO .NET. With Windows Forms, you typically bind sources to a Windows Forms Control. The control then becomes the interface through which you view and modify the data.

Form Data Binding

Providers and consumers of data are required to allow form data binding. It is simpler to look at Windows Forms data binding from the provider perspective. Data binding is versatile in that you can bind to almost any structure that contains data. This can be an array that implements the `IList` interface, a collection, or one of the data structures from ADO .NET. In this section, I will only concentrate on binding with ADO .NET data structures.

 **Note:** The `IList` interface represents a collection of objects that can be individually accessed by index.

You can bind the control to the following ADO .NET data objects:

- **DataColumn object:** This is the building block of a `DataTable` object. It represents a column in a database table. You can bind a simple control, such as a `TextBox` control's `Text` Property, to a column within the data table.
- **DataTable object:** This represents one table of in-memory data in ADO .NET. It can be a one-to-one matching to a database table, or it can be a virtual table derived from the result of a retrieve operation on the database. It contains rows and columns that are represented by two collections, the `DataColumn` and the `DataRow`. You can bind a complex control, such as a `DataGrid` control, to a `DataTable`. However,

when you bind to a `DataTable`, you are really binding to the `DataTable`'s default `DataGridView`.

- **DataView object:** This object is a customized view of a single `DataTable` that may be filtered or sorted. Just like `DataTable`, you can bind `DataGridView` to complex controls, but be aware that you are binding to a fixed snapshot of the data rather than an updating data source.
- **DataSet object:** This is a collection of tables, relationships, and constraints of the data in a database. If you bind to a `DataSet`, you are actually binding to its default `DataGridViewManager`.
- **DataGridViewManager object:** This represents a customized view of the entire `DataSet` and is similar to a `DataGridView` but with relations included.

► **Note:** Simple controls consist mainly of controls that display or hold one element of information. These include controls like text boxes, radio buttons, and check boxes. Complex controls hold a set of elements of information and, at times, even the relationship between the elements. These include grid controls, list boxes, and many other OLE controls.

A `CurrencyManager` object is associated with any Windows Form that you bind to data source. It is the job of the `CurrencyManager` object to keep track of the position in the data source (for example, what row is current) and manage the bindings to the data source. In addition, every Windows Form has a `BindingContext` object. There is a `CurrencyManager` for each discrete data source that you bind to per `BindingContext` object. The `BindingContext` object keeps track of all the `CurrencyManager` objects on the form. So, any Windows Forms with data-bound controls will have at least one `BindingContext` object. You can also create a

BindingContext object for a container control, such as GroupBox, Panel, or TabControl, that contains data-bound controls. This allows each part of your form to be managed by its own CurrencyManager object. Figure 4-1 shows the data binding architecture of Windows Forms.

Common Scenarios for Data Binding

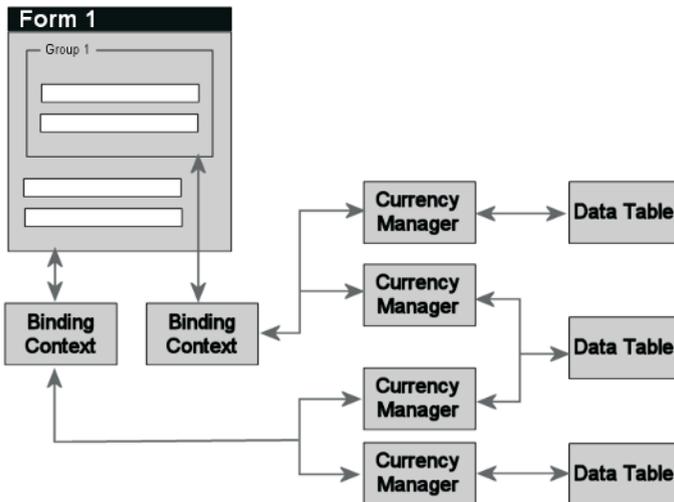


Figure 4-1: Windows Forms data binding architecture

If you take a look at all Windows applications today, you will find that nearly all commercial applications use information read from data sources of one sort or another. Most of those use some kind of data binding technology to display and manipulate the data source. Below are a few of the most common scenarios that use data binding as a method of data presentation and manipulation:

- **Reporting:** Reports provide a flexible way to display and summarize data in printed documents or on screen. Common reports include lists, invoices, summaries, and even cross tabs. The data is formatted to facilitate reading rather than data entry. For example,

you would format the date to display in long date format rather than short date format if space is available.

- **Data entry:** A data entry form is a common way to enter a large amount of related data. Users can enter information directly or select choices using text boxes, option buttons, drop-down lists, and check boxes. The database is updated with the new or modified data.
- **Parent/child relationship:** A parent/child relationship is one format for looking at related data. Typically, there are two tables of data with a relation connecting them (for example, invoice headers and invoice details tables). The relationship is usually one-to-many.
- **Lookup table:** Another common scenario is the table lookup. This is usually a way of finding more details about a row of data. For example, the form will display a list of products sold by a company, but the actual data saved is the primary key of the products table. Since the primary key is just a number and meaningless to a human operator, the name of the item is shown instead.

Data Access Strategy for Windows Forms Applications

There is no correct strategy for accessing data in Windows Forms. Since Windows Forms are typically thick clients and most of the resources consumed are on the client machine, your choice would be mainly in relation to what you expect the client machine to be able to handle. There are a few points you might want to keep in mind:

- **DataSets:** This component allows you to maintain complex relationships and referential integrity, simplifying data manipulation.

- **Data binding:** You can bind data sources to controls in the development environment instead of in the code, simplifying development.
- **Data commands:** You cannot bind to data commands, but some operations that modify database structure can only be done through data commands.
- **Stored procedures:** Creating stored procedures in a database is more efficient than using direct SQL commands to manipulate the data because the stored procedures are compiled.

There is no strategy that fits all situations. Keep in mind the different points mentioned above, and develop your own strategy according to your requirements. You will find that you develop the right strategy as you gain more experience.

Console Applications

Console applications are non-GUI, text-based interfaced applications. Console applications are dated architecture but are still used for some applications. They are very useful when the communication line is slow and processing power is limited. Console applications are widely used for remote administration.

Data Access Strategy for Console Applications

Even though console applications do not have a GUI interface, you might still need to access and process data. You might build a console application that does extensive data processing. You can build an application that carries out data maintenance and administrative tasks such as creating users in the database, or simply allows legacy hardware to access your program.

The data access requirements of a console application are no different from a Windows Forms application. Data is

accessed and processed in the same way. The only thing you cannot do is bind the data to visual components. You will have to write logic to display the data to the user, if required.

Windows Services Applications

Windows Services replaces what was formerly NT Services. A Windows Service is an application or a module that runs on a server and provides services to other applications and modules. A server in this context means the provider of the service. This can include NT Workstation and Windows 2000 Professional, which is not considered a server in the traditional sense. A service has no other user interface. If one is required, a separate module must be written that controls the behavior of the service.

Data Access Strategy for Windows Services

Since there is no direct user interaction, there is no need for data binding in a Windows Service. A Windows Service is typically always running and might service multiple users or multiple processes for the same user. This property puts some unique requirements on Windows Services when it comes to data access.

Since the Windows Service is always running, you might consider not having a permanent connection to the data source, but instead connect to the data source when it is needed. This has the disadvantage of overhead when connecting but frees server resources when the Windows Service is idle. It is also a good idea to check if the connection is live each time you start a data access cycle. Another disadvantage of this strategy is that you can get a lot of connection and disconnection cycles, which is not very efficient.

An improved method is to use a timeout strategy. This means that the connection to the data source disconnects after a set time of being idle. You can then check if a connection is live and only reestablish the connection if it has timed out.

We have so far looked at three types of application architectures, all with similar requirements. Next, we will look at ASP .NET applications, which require more planning and careful attention when it comes to data access strategy.

ASP .NET Web Applications

ASP .NET is not only the next version of Active Server Pages (ASP), but it also provides a unified web development platform for the development of enterprise-class web applications. Although ASP .NET syntax is largely compatible with ASP, it also provides new enhanced features for robust and scalable web applications.

ASP .NET is a compiled, .NET-based environment; you can author applications in any .NET-compatible web language. For now, the languages available are C#, J#, and VB .NET. As ASP .NET is one of the core .NET class libraries, the entire .NET Framework is available to any ASP .NET application, including ADO .NET. This adds the benefits of these technologies to web development, which includes the managed Common Language Runtime environment, type safety, inheritance, object-oriented design, and compiled (instead of interpreted) applications.

ASP .NET makes extensive use of Web Forms and XML Web Services. XML Web Services are used to build Business to Business (B2B) and Business to Client (B2C) applications. You can consider Web Forms to be the presentation tier and Web Services to be the middle tier or business tier of a distributed application.

Web Forms

Web Forms are used to create programmable web pages that provide the user interface for web applications. A Web Form page presents information to the user in any browser or client device and implements application logic using server-side code.

► **Note:** In this context, Web Form page means the web page that is sent to the browser and is generated by the ASP .NET compatible web server from a Web Form.

Previously in ASP, there was no clear separation between code and visual components of the user interface. All code and scripts were included in *.asp files. In ASP, you can use COM to separate business logic, but you still have to have codes that manipulate the visual component in the same file as the component. With ASP .NET, this has now changed. User interface programming of Web Forms is divided into two distinct parts: the visual component in *.aspx files and the logic in *.aspx.vb or *.aspx.cs files.

Data Access in Web Forms

The nature of web programming itself is such that data access in Web Forms differs in several ways from data access in Windows Forms or in older forms technology. You must consider issues such as state management, separation of server and client, designing for scalability, and so on. In addition, because you will be working with databases, you must also understand the important points of how to manage data in Web Forms.

There are a few fundamental principles that you need to bear in mind when accessing data in Web Forms:

- Using a disconnected model
- Reading data more often than updating it
- Minimizing server resource requirements
- Accessing data using remote processes (distributing data access)

Disconnected Model

Web Forms are disconnected. With each request a client makes to the server, the page is built, processed, sent to the client, and discarded from the server memory. As a result, the data on the server are discarded from the server memory along with other elements of the Web Form page.

Data you are working with are not automatically available with each round-trip to the server. If you want to access the data, you must reload it from the source or you must include logic to save and restore the data as part of the page processing. This makes it impractical to maintain database connection. Instead, for each round-trip cycle you need to connect, process data (read or write), and then disconnect from the database.

Reading and Updating

The Web Forms model presumes that most data accessed by pages are read-only. This means that there are more read operations than write operations being performed on the data source. As a result, the Web Forms data binding architecture is one-way. The data binding only displays data in controls but does not write from controls to the data source.

The one-way architecture makes the page more efficient as it removes the bigger overhead that updating requires. If you have a page that requires updating the data source,

you must explicitly write code to perform the update operations yourself.

Minimizing Server Resource Requirements

Since the Web Forms pages are processed on the server before they are sent to the browser, any data access adds additional load to the server resources, both in terms of processing time and memory usage.

If you choose to keep the data on the server between round-trips (e.g., using session state variables to store the data), you use server resources even when the page is not being processed. This might work when you have a small set of users, but it will not allow your application to be scalable to a larger user set.



Tip: When designing Web Form applications, consider the following:

- Be conservative with data retrieval. Only retrieve what you need and no more.
- Use client-side state management to store data if possible.

Accessing Data Remotely

Web Forms are the presentation tier of your web application. Although you can include data access in your page, in a distributed architecture paradigm it is common to separate data access logic and business logic from the user interface logic. This can be achieved by building an XML Web Service that contains the data access logic.

Data to XML Web Services

An *XML Web Service* is a programmable entity that provides a particular set of services or functionality, such as application logic or data access control. It is accessible to any number of systems using ubiquitous Internet standards, such as XML and HTTP. The methods of

communication used by XML Web Services are XML-based messaging. This helps bridge the difference that exists between systems that use incongruent component models, operating systems, and programming languages.

As it is designed to work in the heterogeneity of the web environment, XML Web Services must have the following properties:

- **They must be loosely coupled:** Loosely coupled systems have only the requirement of understanding self-describing, text-based messages to be able to communicate between each other.
- **They must use ubiquitous communication:** The Internet communication capability is now a standard requirement for new operating systems, at least in the near future, thus providing an omnipresent communication channel. The ability to connect almost any system or device to the Internet will ensure such systems and devices are universally available to any other system or device connected to the Internet.
- **They must use universal data format:** Any system supporting the same widely accepted open standards is capable of understanding XML Web Services. By using XML, communication between autonomous and disparate systems is now a possibility.

The DataSet was engineered in such a way to provide convenient transport of data over the Internet. The DataSet and DataTable can be specified as an input or an output of XML Web Services without any additional coding required to stream the contents of the DataSet between the XML Web Services and the client. The DataSet is implicitly converted to an XML stream on the sending end, sent over the network, and reconstructed from the XML stream to a DataSet on the receiving end.

This provides a simple way for XML Web Services to exchange data with its clients. Figure 4-2 shows the XML Web Services communication cycle.

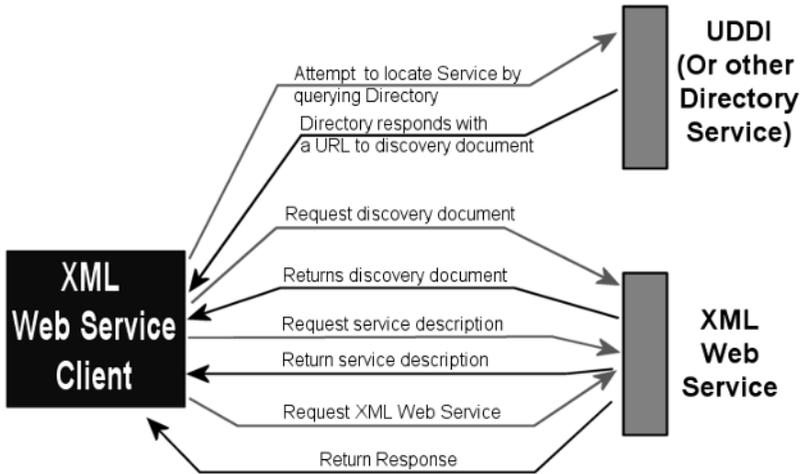


Figure 4-2: XML Web Service communication cycle

► **Note:** The DataSet is converted to an XML stream using the DiffGram format. A DiffGram is an XML format that is used to identify current and original versions of data elements. The DataSet uses the DiffGram format to load and persist its contents and to serialize its contents for transport across a network connection. When a DataSet is written as a DiffGram, it populates the DiffGram with all the necessary information to accurately recreate the contents, though not the schema, of the DataSet, including column values from both the original and current row versions, row error information, and row order.

The DataSet was architected with a disconnected design, in part to facilitate the convenient transport of data over the Internet. The DataSet and DataTable are “serialize-able” in that they can be specified as an input to or output from XML Web Services without any additional coding required to stream the contents of the DataSet

from an XML Web Service to a client and back. The DataSet is implicitly converted to an XML stream using the DiffGram format, sent over the network, and reconstructed from the XML stream as a DataSet on the receiving end. This gives you a very simple and flexible method for transmitting and returning relational data using XML Web Services.

The code sample below shows a simple use of the DataSet in an XML Web Service.

The first thing you need to do is create the XML Web Service. We will first create a Class called Service1 that will manipulate employee data in the Northwind database. Once you have generated a template for your services, you will add the following code to the top of the Service1.asmx.vb file:

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Web.Services

<WebService(Namespace="http://localhost/CH07- _
Sample-01")> Public Class Service1
    Inherits System.Web.Services.WebService
```

You will place the rest of the code after the Web Services designer generated code, still in Service1.asmx.vb file:

```
'WEB SERVICE SAMPLE

'Create SQL Connection to database
Public nwindConn As SqlConnection =
    New SqlConnection ("Data Source=localhost; _
        Integrated Security=SSPI;
        Initial Catalog=northwind")

'Describe public function for getting employee data
<WebMethod(Description:="Returns Northwind
employee", EnableSession:=False)> _
    Public Function GetEmployee() As DataSet
```

```

'Declare and initialize SQL DataAdapter
Dim employeeDA As SqlDataAdapter =
    New SqlDataAdapter ("SELECT EmployeeID,
        LastName, FirstName, Title FROM Employees",
        nwindConn)

'Declare and initialize DataSet
Dim employeeDS As DataSet = New DataSet()

'Determine action to take if column name does not
'match
employeeDA.MissingSchemaAction = _
    MissingSchemaAction.AddWithKey
employeeDA.Fill(employeeDS, "Employees")

GetEmployee = employeeDS

End Function

'Public function for modifying Employee table
<WebMethod(Description:="Updates Northwind
Customers", EnableSession:=False)>
Public Function UpdateEmployee(ByVal employeeDS As
DataSet) As DataSet

    'Create SQL DataAdapter
    Dim employeeDA As SqlDataAdapter = New
        SqlDataAdapter()

    'Define the insert command
    employeeDA.InsertCommand = New SqlCommand
        ("INSERT INTO Employees (EmployeeID, LastName,
        FirstName)" & "Values(@EmployeeID, @LastName,
        @FirstName)", nwindConn)

    employeeDA.InsertCommand.Parameters.Add _
        ("@EmployeeID", SqlDbType.NChar, 5, "EmployeeID")

    employeeDA.InsertCommand.Parameters.Add _
        ("@LastName", SqlDbType.NChar, 15, "LastName")

    employeeDA.InsertCommand.Parameters.Add _
        ("@FirstName", SqlDbType.NChar, 15, "FirstName")

    'Define the update command

```

```

employeeDA.UpdateCommand = New SqlCommand _
("UPDATE Employees Set LastName = @LastName, " & _
"FirstName = @FirstName WHERE EmployeeID = _
@EmployeeID", nwindConn)

employeeDA.UpdateCommand.Parameters.Add _
("@LastName", SqlDbType.NChar, 15, "LastName")

employeeDA.UpdateCommand.Parameters.Add _
("@FirstName", SqlDbType.NChar, 15, "FirstName")

'Define the where clause parameter as that of
'the original employee ID
Dim myParm As SqlParameter = _
employeeDA.UpdateCommand.Parameters.Add _
("@EmployeeID", SqlDbType.NChar, 5, "EmployeeID")

myParm.SourceVersion = DataRowVersion.Original

'Define the Delete command
employeeDA.DeleteCommand = New SqlCommand _
("DELETE FROM Employees WHERE EmployeeID = _
@EmployeeID", nwindConn)

'Define the where clause parameter as that
'of the original employee ID
myParm = employeeDA.DeleteCommand.Parameters.Add _
("@EmployeeID", SqlDbType.NChar, 5, "EmployeeID")

myParm.SourceVersion = DataRowVersion.Original

employeeDA.Update(employeeDS, "Employees")

UpdateEmployee = employeeDS

End Function

End Class

```

After you have created the XML Web Service, you can test it by running the debugger in Visual Studio .NET. This should open a web page with the hyperlinks for the two methods you have just created: GetEmployee and UpdateEmployee. If you click on GetEmployee, a new

web page opens containing a button called Invoke. If you click on Invoke, you get the XML result of running the GetEmployee method. There is no such button for the UpdateEmployee method. This is because it requires parameters, and to test it, you will need to define debug data for the method.

Once you have created the required web reference in your client application, you can access the methods as if it were any other local object. In the listing below, a web reference has been defined to the XML Web Service we created above; it is called ServiceSample.

```
'Define Object from XML WebService
Dim ServiceClient As New ServiceSample.Service1()

'Get the DataSet using the GetEmployee method
Dim myDS As DataSet = ServiceClient.GetEmployee

'Get table
Dim myTable As DataTable = myDS.Tables("Employee")
```

As you can see, once you have created the web reference, it is a simple matter of creating an object based on the class in the XML Web Service. Communication and getting results from methods of the object are handled transparently, just as if the object were local.

Data Access Strategy for ASP .NET Applications

When you design your web applications, you will need to decide what data access strategy you wish to adopt. There is no right strategy; each one has its own advantages and disadvantages that you must first consider. You will have to make a choice as to which strategy you adopt, depending on your particular requirements.

DataSets or Data Commands?

One of the first choices you need to make is whether to cache data in DataSets or access the database directly

reading rows through a data reader. For some database operations, that results in modification of the database structure (for example, creating new tables—you cannot use DataSets, but you have to execute a data command instead). For most common data access scenarios, however, you have a choice between storing records in disconnected DataSets and accessing the records directly using data commands.

Each strategy has inherent advantages that apply to any data access scenarios and not just for web applications. Using DataSets makes it easier to work with related tables and data from disparate sources. On the other hand, using a data reader eliminates the extra steps of filling a DataSet. This often results in slightly better performance and memory usage. You also have more direct control over the statements and stored procedures you use.

DataSets and Data Commands in Web Forms Pages

When using Web Forms, additional factors come into play when choosing your data access strategy. One main factor is the Web Form life cycle; Web Forms are initialized, processed, sent to the client, and discarded with each round-trip to the server. If you just want to display data, using a DataSet is inefficient and requires unnecessary overhead since that DataSet will immediately be discarded.

In general, you can assume that using data commands is better when working with Web Forms pages. However, there are exceptions:

- **Working with related tables:** With DataSets, you can maintain multiple related tables, including support for relations and referential integrity. When you work with related records, such as parent and child relationships, it can be much simpler to use a

DataSet rather than fetching the records independently using data commands.

- **Exchanging data with other processes:** If you exchange data with other components, such as XML Web Services, you will almost always use a DataSet to hold a local copy of the data. As discussed earlier, DataSets automatically read and write the XML stream used to communicate between components in the .NET Framework.
- **Working with a static set of records:** If you use the same set of records repeatedly, such as paging in a grid, it is more efficient to place those records into a DataSet rather than retrieving the data from the database with each round-trip.



Tip: Remember to always retrieve, whenever practical, only the records and columns that you need and no more. This will reduce load on server resources and make your application more scalable.

Cache or Recreate?

If you choose to use DataSets, your next choice is to decide whether to recreate the DataSet with each round-trip or create it once and save it in such a way that it can be accessed in a subsequent round-trip.

If you choose to recreate the DataSet with each round-trip, you have to run a query against the database each time a user clicks a button on your page. The advantage is there is less chance of the data being out of sync, but there is the added overhead of connecting to the database each time.

If you save and restore the DataSet with each round-trip, you reduce the overhead of connecting to the server but increase the load on server resources. If the recordset is large, and you have a lot of users, you can quickly run out

of server resources (namely, memory). You can, however, store the DataSet on the client, as I will discuss in the next section. There is also a bigger chance of the data being out of sync since you are not refreshing the data with each round-trip.

Do You Store the DataSet on the Server or Client?

If you choose DataSets, the final decision is where to store the DataSet. You can store it on the server as a session variable or application variable, or you can store it in the client page in a hidden field. If you store it on the server, you will of course use server resources. This makes the application less scalable. Conversely, if you store the DataSet in the page, it will be passed as part of the HTML stream to the client. If the DataSet is large, the communication speed between server and client can be adversely affected.

No matter which strategy you choose, you will have to write the logic yourself for storing the DataSet on Web Forms. DataSets are stored as type Object in session variables, and you must cast it back as DataSet. See the following example:

```
Private Sub Page_Load(ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) _
    Handles MyBase.Load
    ' Check to see if the page is loaded again
    If Page.IsPostBack Then
        ' Cast object as dsEmployees from session
        ' and assign to variable
        dsEmployeeLD = CType(Session("myDsEmployees"), _
            dsEmployees)
    Else
        ' If this is first time page is loaded check
        ' session variable
        If Session("myDsEmployees ") Is Nothing Then
            ' Variable does not exist, create it
            ' and set its value
            OleDbDataAdapter1.Fill(dsEmployeeLD)
            Session("myDsEmployees ") = dsEmployeeLD
        End If
    End If
End Sub
```

```
End If  
End If  
End Sub
```

Concurrency Issues

In a multiuser and distributed environment, there is often the risk that two users will update the same records or the records will be out of sync. This is common to a concurrent system. There are two strategies that can be adopted to overcome concurrency issues: pessimistic concurrency and optimistic concurrency.

Pessimistic concurrency involves locking rows of records while the user is working on them. It means that no other user can update the records while one user is working on them. This strategy is primarily used in an environment where there is heavy contention for data. It is not really appropriate for web applications since the connection to each client is not maintained with each round-trip. Once a connection closes, all locks that it holds are automatically released.

Optimistic concurrency does not lock rows. Instead, it checks if the row has changed since it was last read before applying any update. If it has not been changed, the update can proceed as normal; otherwise, the user is informed about the change and given a choice to re-retrieve the data and discard changes or overwrite the changed record. The DataSet object is designed to encourage the use of optimistic concurrency for long-running processes, such as those found in distributed applications and web applications.

A common method to determine if the records have changed is to check each field against what was originally retrieved. This, though more accurate, will add additional overhead since the number of fields you have to compare can be potentially large. The more practical method is to design optimistic concurrency checking within your

database and application. This can be done by having a date and time field with every updateable table. Each time the table is modified, the current date and time is set in the table. This can be achieved through database triggers or done by the application itself. You will then only have to check the date and time field to know if the record has changed since you last retrieved it.

Data, Data Everywhere

We have so far examined all the different application models available in .NET, different strategies you can follow when manipulating data, and what the different implications are. As you have seen, there is not always a clear-cut solution or correct strategy. You will have to weigh the advantages and disadvantages of each one before you can choose which strategy is appropriate for your application.

Spec My Components

Previously in this chapter, you learned about the different application architectures and how ADO .NET is integrated in each one. We covered ASP .NET in more detail because its architecture is substantially different for classical applications. ASP .NET is based more on distributed application architecture. Distributed application has at least three main layers, or tiers: the presentation tier, the business tier, and the data tier. Components in any of the tiers, especially the business and data tiers, are designed to be independent and can be used as building blocks to different applications. In the following sections we will look at data components, which are found in the data tier.

What is a Component in .NET?

Generally in programming, “component” is used to refer to self-contained, compiled pieces of code that are reusable and can interact with other codes and objects. Previously, components were implemented through the COM model or derivatives, such as COM+ and DCOM. In the .NET Framework, a component is simply a class that implements the `System.ComponentModel.IComponent` interface or is inherited directly or indirectly from a class that implements `IComponent`. A .NET Framework component also provides additional features, such as control over external resources and design-time support.

Components are hosted or sited within a container.

- **Container:** This is a class that implements the `System.ComponentModel.IContainer` interface or is inherited from a class that does. The `IContainer` interface must support methods for adding, removing, and retrieving components. A container contains one or more components that are referred to as the container’s child components.
- **Site:** This is a class that implements the `System.ComponentModel.ISite` interface or is inherited from a class that does. Sites are provided by a container as a way to manage and communicate with their child components. Typically, a container and a site are implemented as one unit.

When to Build Data Components

When you develop an application that shares data, you should consider merging that data into a single component. If you store your data via data components, you provide a standard way for other components to access the data. For example, if you have a stock entry component, it can be used by both an inventory management component and an order processing component. You first

have to determine the component's functionality, and only then will you be prepared to refine the component design and implementation.

Component Design Guidelines

When you design your component, your first decision is which tier the component will fall into. This is important, but it is not always easy to separate the logic. It takes some experience to correctly do the separation, and it will affect how you design the component. In this book, we are more interested in the data tier, so I will concentrate on design guidelines for data components, especially from the point of view of ASP .NET applications.

All your component design decisions are interrelated. Your component function will affect how it is used in your applications and will, in turn, affect what kind of models you will use.

Component Scope

When you create an instance of a component in your ASP .NET application, you must decide what its scope will be. You have three scopes available to you:

- **Page:** Most business rule components are placed at the page scope. The object created on the page scope is available on the page. It is created with each round-trip and destroyed each time.
- **Session:** If the component functionality spans multiple pages, it should be at the session scope. Session scope objects are created for each session a user has with the server. The object is only destroyed when the user disconnects from the server. This is the usual scope for data components. Be aware that session variables consume resources on the server, and this can have some issues with scalability.

- **Application:** This scope is usually reserved for application-wide components, such as a page counter component. Variables and objects at application level are available to all users and sessions.

The scope of your component will determine which methods you choose to implement in your component and how you store state variables. If you have a component that is scoped for page, there is no need to store data in variables so that the successive page accesses them. Remember, the component is destroyed in each round-trip. Instead, you would have to store the data in the data source directly and recreate them in each round-trip or each time the component is constructed.

Component Implementation

Implementing components in .NET is as simple as implementing any other kind of class with a few rules that you must follow so that the class is considered to be a component, a container, or a site, as we discussed previously. There are also a few guidelines that you are advised to follow so that your component is efficient, maintainable, and scalable. We will discuss issues specific to the implementation of components next.

Properties vs. Public Fields

In a class, you can give the developer access to member fields by either making the fields public or providing methods that modify those fields. To maintain the rule of encapsulation, you should avoid the use of public fields in all your classes. In components, methods that modify fields have a special syntax and are called properties.

Visual designers, such as Visual Studio .NET, display properties but do not display public fields. Therefore, it is better for a component to define properties instead of public fields. Public fields work against the principle of encapsulation in the object-oriented paradigm, whereas

properties embrace encapsulation. Properties act like intelligent fields and normally have a private data member combined with modifier and accessor functions. It is then accessed syntactically as a field of a class.

A property definition generally consists of the following two pieces: a private or protected data field and a public or protected property.

```
Public Class APropertyExample

    ' The data field.
    Private amount As Integer = 0

    Public Property propAmount As Integer
        ' Retrieves amount.
        Get
            Return amount
        End Get

        ' Assigns to amount.
        Set
            amount = value
        End Set
    End Property

    'Other members...
End Class

' Example of how to use the property
Public Class UseAPropertyExample
    Public Shared Sub Main()

        ' The data field.
        Dim example As New APropertyExample()

        ' Sets the property.
        example.propAmount = 5

        ' Gets the property.
        Dim anumber As Integer = example.propAmount
```

```
End Sub
```

```
End Class
```

As you can see in the code snippet above, the property is further divided into two parts: the data accessor (or Get function) and the data modifier (or Set function).

Learning to Run

Once you have built your application, you usually find that you need to optimize it so it can scale better and perform as well as it did in the production environment. In the following sections, you will learn a few additional techniques for improving data access.

Connection Pooling

Pooling is a way to share data connections in applications without the need and overhead of establishing a connection each time. Pooling connections can significantly enhance the performance and scalability of your applications. Most database vendors provide ways to manipulate or create connection pools. I will concentrate mainly on the data provider available with .NET, specifically the SQL Server .NET Data Provider. The SQL Server .NET Data Provider provides connection pooling automatically for ADO .NET client applications that connect to any SQL Server database. You can also supply several connection string modifiers to control connection pooling behavior.

When you open a connection, a connection pool is automatically created based on an exact string matching algorithm that associates the pool with the connection string in the connection. Each connection pool is associated with a distinct connection string. This means that if you already have a connection with the same connection

string as you have just specified, that connection will be used instead of having to create a new one. If an exact match to an existing pool is not found, a new connection is created and added to the pool.

In the code snippet below, three new `SqlConnection` objects are created, but only two connection pools are required to manage them.

```
SqlConnection conn = new SqlConnection()
conn.ConnectionString = _
    "Integrated Security=SSPI;Initial Catalog=northwind"
connA.Open()
' Pool A is created.

SqlConnection conn = new SqlConnection()
conn.ConnectionString = _
    "Integrated Security=SSPI;Initial Catalog=pubs"
conn.Open()
' Pool B is created because the connection strings differ.

SqlConnection conn = new SqlConnection();
conn.ConnectionString = _
    "Integrated Security=SSPI;Initial Catalog=northwind"
conn.Open()
' The connection string matches pool A
' so A is used instead of creating a new one.
```

Once created, connection pools are not destroyed until the active process ends. Maintenance of inactive or empty pools involves minimal system overhead.

When a pool is created, multiple connection objects are created and added to the pool. The number of connections created are based on the minimum pool size requirements. Connections are then added to the pool as and when required and up to the maximum pool size. Minimum and maximum pool size are specified as part of the connection string.

When an `SqlConnection` object is requested, it is obtained from the pool if a usable connection is available. To be usable, the connection must currently be unused,

have a matching transaction context or not be associated with any transaction context, and have a valid link to the server.

When the maximum pool size is reached and no usable connection is available, any additional request is queued until a connection does become available or the timeout period is over, in which case an error occurs.



Warning: Connections that are not explicitly closed are not added or returned to the pool. You must always close the connection yourself when you are finished using it. This can be done by using either the `Close` or `Dispose` method.

Stored Procedure or SQL Statement?

Stored procedures are generally faster than their SQL statement equivalents. The main reason is that stored procedures are compiled and an execution plan is worked out at design time. On the other hand, SQL statements have to be interpreted by the database engine, and the execution plan has to be worked out at run time, which adds additional overhead. Generally, you should try to manipulate the database through stored procedures instead of SQL statements.

Which Data Type?

When designing your database, one of the most important considerations is the use of data types. Data types are not all created equal, even if they are sometimes interchangeable. For example, if you create a `varchar(10)` data type, you could also create a `char(10)` data type instead. However, `varchar` data types, though more efficient in storage, have added overhead for maintenance. If all the data is going to be ten characters long, `char(10)` is more efficient; on the other hand, if the data varies in length, `varchar(10)` is more efficient.

The idea here is for you to be familiar with the data type available in your database and choose wisely according to your needs and future anticipated needs.

Data Warehousing

Data warehousing is a huge topic in itself, but it is mentioned here as a way to optimize your online transaction processing (OLTP) database. As time goes by, most OLTP databases accumulate historical data that are not required for the day-to-day running of the system but are kept because they are required for statistical analysis. When you use data warehousing, you regularly clean the system, keeping it small and optimal while you maintain the data required for analysis in a data warehouse. The data warehouse can then be used as a source for an online analytical processing (OLAP) system, which is more appropriate for larger volume statistical analysis.

Tuning and Monitoring

Once the system goes live, it is imperative that you continually monitor and tune the application to match your needs. Initially, that might involve redesigning certain parts of the system, but as the system stabilizes, these issues can be solved by tuning the environment itself, such as the server or network, or even getting more powerful hardware. Your aim should be to identify bottlenecks and make the system causing these bottlenecks as efficient as possible.

To help with monitoring and identifying bottlenecks, many servers (Windows 2000, MS SQL 2000, and Internet Information Services) include performance counters and monitors that you can use. Some, like MS SQL Server, even include a self-tuning option. Study the different performance counters that are available on your system and make good use of them.

Protecting the Application

Since applications are becoming more distributed and the use of the Internet has exploded, it is apparent that security even to the level of data access is paramount and can no longer be taken for granted. In the remainder of this chapter, we will look at some security issues relating to data access.

Passwords, Users, and Access Rights

To access a database, especially if you are designing a component or XML Web Service, you will need to have an account that the service can access. This is because the service is more or less public. Clients to the service will have no idea which user name or password they should use, and you should not try to set one for each client, especially considering that the potential number of clients can be huge. You will have to set one account that the service can use to access data on the database.

X Warning: To protect your database, you must never use a hard-coded password. With the .NET Framework, there are management tools such as ILDASM.EXE, which parses .NET Framework EXEs and DLLs and shows readable information about the files. If you have a hard-coded password, it will be exposed.

The security mechanism in .NET has been enhanced from previous versions of Windows. The classic object-based security (files, services) still exists, but now there is also code-based and evidence-based security. That means that a piece of code can be given security profiles defining which action it is allowed to perform. An administrator can also gather evidence to identify who and where a piece of code comes from. When designing your application, you will have to keep this in mind.

Application Information

Currently on the Windows platform, most application information resides in the Windows registry. The .NET Framework was designed for the web and, as such, should be as platform independent as possible. The best practice in .NET suggests that application information reside in regular files instead of the registry.

Summary

In this chapter you learned about the different kinds of application models available in .NET and how ADO .NET fits into each. You learned about the difference between classic Windows applications, web applications, and XML Web Services. You also learned about the different strategies that you can use in each model to manipulate data. There is no hard-and-fast rule as to which strategy is better. Each has advantages and disadvantages, and it all comes down to your experience to choose the best one for the job at hand.

XML Integration with ADO .NET

XML in .NET Frameworks

XML stands for Extensible Markup Language and was developed by the World Wide Web Consortium (W3C). XML was designed mainly to overcome the limitation of HTML. Microsoft has embraced XML, and it plays a major part in the .NET Framework.

In the previous chapter, we saw how XML is transparently used for communication between XML Web Services. In this chapter, we will learn more about how XML fits in the .NET Framework in general, and we will go in more detail about the integration of XML in ADO .NET.

Architectural Overview and Design Goals

XML integration in .NET Framework was designed to meet certain goals:

- Compliance with the W3C standards
- Extensibility
- Pluggable architecture
- Performance
- Tight integration with ADO .NET

Standards Compliance

.NET fully conforms to the W3C recommended standards of XML, Namespaces, XSLT, XPath, Schema, and the Document Object Model (DOM). Compliance is essential to ensure interoperability across platforms.

- **XSLT:** Extensible Stylesheet Language (XSL) Transformation is used to transform the content of a source XML document into a presentation that is tailored specifically to a particular user, media, or client.
- **XPath:** XPath is a query language used for addressing parts of an XML document.

.NET Framework contains sets of XML classes that support the W3C XML Schema Definition (XSD) language 1.0 recommendation.

Extensibility

Extensibility is achieved through the use of abstract base classes and virtual methods. This extensibility is also referred to as *subclassing* and is illustrated by the `XmlReader`, `XmlWriter`, and `XPathNavigator` abstract classes. These classes enable new implementations to be developed over different data sources and stores, exposing them as XML. The existing data source and stores can include any file systems, registries, flat file legacy databases, and relational databases. The new implementations not only display the data as XML but also provide XPath query support for those stores.

Pluggable Architecture

XML in the .NET Framework is a stream-based architecture. Pluggable in this architecture means that components that are based on abstract .NET XML classes can easily be substituted. It also means that if you have data streaming between the components, new components

inserted or plugged into the stream can alter the processing. For example, you can plug components together using different data stores, such as an XPathDocument and XmlDocument in the transformation process. You could plug an implementation of your own XmlReader or XmlWriter for processing the output, allowing the transformation process to and from virtually any data source. To allow the processing of a new data source, simply implement your own XmlReader or XmlWriter for that data source and plug it in.

Performance

XML classes in .NET Framework represent low-level processing components and are required to have high performance. They are designed to support a streaming-based architecture. For improved performance, they have the following characteristics:

- Minimal caching for forward-only, pull model parsing with the XmlReader
- Forward-only validation with the XmlValidatingReader
- Cursor style navigation of the XPathNavigator, which minimizes node creation to a single virtual node, yet provides random access to the document. It does not require a complete node tree to be built in memory like the DOM.
- Incremental streaming output from the XslTransform class

Tight Integration with ADO .NET

In the .NET Framework, relational data and XML are coupled through tight integration between the XML classes and ADO .NET. The DataSet component in ADO .NET has the ability to read and write XML using XmlReader and XmlWriter classes, including the ability

to persist its relational schema as XML Schemas and construe the schema structure from an XML document. DataSet and XmlDataDocument can be synchronized so that changes in one can be reflected in the other. We will learn more about XML integration with ADO .NET later in this chapter.

DOM: The XML Document Object Model

The Document Object Model (DOM) class is simply an in-memory representation of an XML document, which allows you to programmatically read, manipulate, and modify XML documents. In .NET, the DOM is presented by the XmlDocument object. Editing is the primary function of the DOM. It is the structured way that XML data is represented in memory, even though the actual XML data is stored in a linear fashion when in a file or in an XML stream from another object.

The DOM is represented as a tree. The basic element of the DOM tree is a node, which is represented in .NET by an XmlNode object. Consider the following XML data.

```
<?xml version="1.0"?>
  <products>

    <product>
      <productname>Smelly Cheese</productname >
      <price format="dollar">100.99</price>
      <expirydate>01/01/2009</expirydate>
    </product>

    <supplierinfo>
      <supplier>Good Cheese Express</supplier>
      <state>WA</state>
    </supplierinfo>

  </products>
```

The illustration below shows the DOM tree for the XML data:

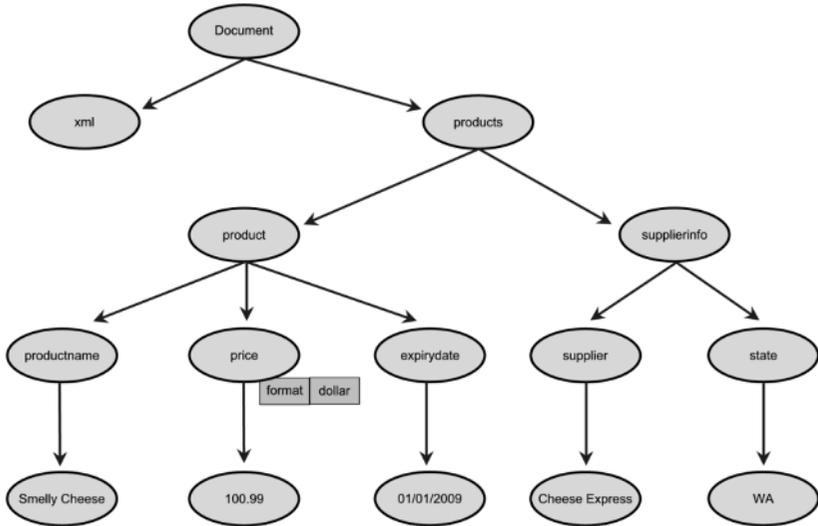


Figure 5-1: The DOM tree

In the illustration, each circle represents a node. Node objects have set methods and properties, as well as some basic characteristics:

- Nodes have a single parent and most can have multiple child nodes.
- There are different types of nodes that can have multiple child nodes:
 - Document
 - DocumentFragment
 - EntityReference
 - Element
 - Attribute
- There are a few types of nodes that cannot have child nodes:

- XmlDeclaration
 - Notation
 - Entity
 - CDATASection
 - Text
 - Comment
 - ProcessingInstruction
 - DocumentType
- Attribute is one special node that does not have siblings, parent, or child.

► **Note:** Attributes, although defined as nodes by the WC3 standards, are better considered a property of an element node. Attributes are made up of a name and value pair (for example, format="dollar").

Nodes on the same level in the DOM tree are siblings, such as with the product node and the supplierinfo node in Figure 5-1.

The XmlDocument class extends the XmlNode and supports methods for performing operations on the document as a whole, such as, loading into memory or saving the XML to a file. In addition, XmlDocument provides a means to view and manipulate the nodes in the entire XML document.

► **Note:** For optimization purposes, if you do not require the structure or editing capabilities provided by the XmlDocument class, the XmlReader and XmlWriter classes provide non-cached, forward-only stream access to XML.

Nodes in .NET

Since a node is the basic structure for the DOM, let's look at the different node types that .NET supports in more detail.

W3C DOM Node Type	.NET Class	Description
Document	XmlDocument	The container of all the nodes in the tree. It is also known as the document root, which is not always the same as the root element.
Document-Fragment	XmlDocument-Fragment	A temporary bag containing one or more nodes without any tree structure
DocumentType	XmlDocumentType	Represents the <!DOCTYPE...> node
EntityReference	XmlEntityReference	Represents the non-expanded entity reference text
Element	XmlElement	Represents an element node
Attr	XmlAttribute	An attribute of an element accessed using the GetAttribute method of an XmlElement
Processing-Instruction	XmlProcessing-Instruction	A processing instruction node
Comment	XmlComment	A comment node
Text	XmlText	Text belonging to an element or attribute
CDATASection	XmlCDATASection	Represents CDATA
Entity	XmlEntity	Represents the <!ENTITY...> declarations in an XML document, either from an internal document type definition (DTD) subset or from external DTDs and parameter entities
Notation	XmlNotation	Represents a notation declared in the DTD

W3C DOM Node Type	.NET Class	Description
Not in W3C specification	XmlDeclaration	Represents the declaration node <?xml version="1.0"...>
Not in W3C specification	XmlSignificant-Whitespace	Represents significant white space, which is white space in mixed content
Not in W3C specification	XmlWhitespace	Represents the white space in the content of an element
Not in W3C specification	EndElement (not a class)	Returned when XmlReader gets to the end of an element (for example, XML: </item>)
Not in W3C specification	EndEntity (not a class)	Returned when XmlReader gets to the end of the entity replacement as a result of a call to ResolveEntity

Loading XML Documents in the DOM

XML information is read into memory from different formats or sources. These can be a stream, URL, text reader, XmlReader object, or derived class of the reader. The Load method loads the document into memory. It is an overloaded method that can take data from each of the different formats. There is also a LoadXml method that reads XML from a string, which is the method we will be using in the following example.

```
Imports System
Imports System.IO
Imports System.Xml

Public Class Sample

    Public Shared Sub Main()
        'Create the XmlDocument.
        Dim doc As New XmlDocument()
        Dim XmlString As String
```

```

'Define the XmlString
XmlString = _
    "<?xml version=""1.0""?>" & _
    "<products>" & _
    "<product>" & _
    "<productname>Smelly Cheese</productname>" & _
    "<price format=""dollar"">100.99</price>" & _
    "<expirydate>01/01/2009</expirydate>" & _
    "</product>" & _
    "<supplierinfo>" & _
    "<supplier>Good Cheese Express</supplier>" & _
    "<state>WA</state>" & _
    "</supplierinfo>" & _
    "</products>"

'Load the DOM
doc.LoadXml(XmlString)

'Save the document to a file.
doc.Save("Smelly Cheese data.xml")

End Sub 'Main

End Class Sample

```

The above example does not do much; it just creates the DOM for a string and saves it to a file. Notice that you need the System.Xml namespace to be able to use XML classes and System.IO to save the file.

Validating XML Documents

Schemas are used to validate XML documents to make sure that they are well-formed and follow certain required rules. XML documents can be validated using a document type declaration (DTD) file or an XML Schema.

DTD: The XML Document Type Declaration

The document type declaration is used to validate XML documents. It is the original schema definition language for XML. DTDs have their own syntax and rules, which are different from XML. In XML documents, the `<!DOCTYPE>` statement is used to link the document to a DTD. DTDs are somewhat limited when compared to the more flexible XML Schema.

In .NET, the `XmlValidatingReader` class is used to validate an XML document against an inline DTD section or an external DTD file. To perform validation against a document type definition, `XmlValidatingReader` uses the DTD defined in the `DOCTYPE` declaration of an XML document. The `DOCTYPE` declaration can either point to an inline DTD or be a reference to an external DTD file.

SOM: The XML Schema Object Model

The Schema Object Model (SOM) classes provide an in-memory representation of an XML Schema, which allows you to create and validate XML documents. XML Schemas are similar to data modeling in a relational database in that they provide a way to define the structure of XML documents. This is achieved by specifying the elements that can be used in the documents, including the structure and types that these elements must follow. The schema itself is an XML file, typically with an `.xsd` file extension. XML Schemas provide some advantages over document type definitions:

- Additional data types
- Ability to create custom data types
- Schema uses XML syntax
- Schema supports object-oriented concepts like polymorphism and inheritance

In .NET, SOM facilities are provided by a set of classes in the System.XML.Schema namespace.

The World Wide Web Consortium (W3C) schema recommendation specifies the data types that can be used in XML Schemas. In .NET, these data types are represented as XmlSchemaDatatype objects. An XmlSchemaDatatype object contains the ValueType property, which holds the name of the type, as specified in the W3C XML 1.0 recommendation, and the TokenizedType property, which holds the name of the equivalent .NET data type. The table below shows the equivalent .NET data type for each XML Schema data type:

XML Schema Data Type	.NET Framework Data Type
anyURI	System.Uri
base64Binary	System.Byte[]
Boolean	System.Boolean
Byte	System.SByte
Date	System.DateTime
dateTime	System.DateTime
decimal	System.Decimal
Double	System.Double
duration	System.TimeSpan
ENTITIES	System.String[]
ENTITY	System.String
Float	System.Single
gDay	System.DateTime
gMonthDay	System.DateTime
gYear	System.DateTime
gYearMonth	System.DateTime
hexBinary	System.Byte[]
ID	System.String
IDREF	System.String
IDREFS	System.String[]
int	System.Int32

XML Schema Data Type	.NET Framework Data Type
integer	System.Decimal
language	System.String
long	System.Int64
month	System.DateTime
Name	System.String
NCName	System.String
negativeInteger	System.Decimal
NMTOKEN	System.String
NMTOKENS	System.String[]
nonNegativeInteger	System.Decimal
nonPositiveInteger	System.Decimal
normalizedString	System.String
NOTATION	System.String
positiveInteger	System.Decimal
QName	System.Xml.XmlQualifiedName
short	System.Int16
string	System.String
time	System.DateTime
timePeriod	System.DateTime
token	System.String
unsignedByte	System.Byte
unsignedInt	System.UInt32
unsignedLong	System.UInt64
unsignedShort	System.UInt16

The `XmlSchemaElement` and `XmlSchemaAttribute` classes both have `AttributeType` properties and `ElementType` properties that contain an `XmlSchemaDatatype` object once the schema has been validated and compiled.

XML Integration with Relational Data

With previous versions of ActiveX Data Objects (ADO), code written to work with relational data was different from code written to work with hierarchical data. This meant that you had two programming models to work with. In the .NET Framework, several classes in XML are integrated with classes in the ADO .NET architecture, unifying the two programming models. In .NET, the DataSet represents a relational data source in ADO .NET, whereas the XmlDocument implements the DOM in XML. The XmlDataDocument unifies the ADO .NET and XML by representing relational data from a DataSet and synchronizing it with the XML document model.

XML with MS SQL Server 2000

Before we move on, it is interesting to note that Microsoft SQL Server can also directly return result sets as XML. The following example returns all rows from the Employees table from the Northwind sample database encoded as nested XML:

```
SELECT * FROM Employees FOR XML AUTO, ELEMENTS
```

The partial result will be something like this:

```
<Employees>
  <EmployeeID>1</EmployeeID>
  <LastName>Davolio</LastName>
  <FirstName>Nancy</FirstName>
  <Title>Sales Representative</Title>
  <TitleOfCourtesy>Ms.</TitleOfCourtesy>
  <BirthDate>1948-12-08T00:00:00</BirthDate>
  <HireDate>1992-05-01T00:00:00</HireDate>
  <Address> 507 - 20th Ave. E. Apt. 2A</Address>
  ...

```

You can make use of this feature in .NET if you wish:

```
Dim custXmlCMD As SqlCommand
'Define SQL Command that returns XML
custXmlCMD = _
    New SqlCommand("SELECT * FROM Customers FOR XML AUTO,
                    ELEMENTS", northwindconnection)

'Execute the command directly into an XmlReader
Dim myXmlReader As System.Xml.XmlReader = _
    custXmlCMD.ExecuteXmlReader()
```

Expect to see future versions of Microsoft SQL Server to support more XML functionality. The next version of MS SQL, code-named Yukon, will be integrated in the .NET Framework. The stored procedures will run on the .NET Common Language Runtime (CLR). In effect, all the .NET XML features will also be available in Yukon.

DataSet and XML

Using ADO .NET, you can fill a DataSet from an XML data source. The XML data source can be an XML stream or document. You can use an XML data source to supply the DataSet with data, schema information, or both. You can use the information supplied and combine it with existing data or schema in the DataSet.

ADO .NET also allows you to do the reverse. With a DataSet, you can create its XML representation, with or without its schema, in order to transport the DataSet across HTTP for use on other XML-enabled platforms. In the generated XML representation of the DataSet, data is written in XML and the schema is written using the XML Schema Definition (XSD) language, if it is included inline in the representation. This provides a convenient format for transferring DataSet contents to and from remote clients over standard existing Internet HTTP infrastructures.

DiffGrams

The DataSet uses the DiffGram XML format to keep track of changes in XML data. This is particularly important in a stateless web environment where it is not practical to maintain a continuous connection to a database. The DiffGram format is used to identify current and original versions of data elements. The DataSet uses the DiffGram format to load, persist, and serialize its contents for transport across a network connection. The DataSet populates the DiffGram with all necessary information to accurately recreate the contents, though not the schema, of the DataSet. The DiffGram includes column values from both the original and the current row versions, row error information, and row order.

The DiffGram format that is used by the .NET Framework can be used as a basis for communication with other platforms. When sending and retrieving a DataSet from an XML web service, the DiffGram format is implicitly used, even though this is more or less transparent to the developer. With the ReadXml and WriteXml method, you can explicitly specify that the content to be read is a DiffGram or the content is to be written as a DiffGram.

The DiffGram format is divided into three sections: the current data, the original data, and an errors section.

```
<?xml version="1.0"?>
<diffgr:diffgram
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
  xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <Data>
    ...
  </Data>

  <diffgr:before>
  </diffgr:before>
```

```
<diffgr:errors>
</diffgr:errors>
</diffgr:diffgram>
```

The DiffGram format consists of the following blocks of data:

<Data> </Data>: The name of this element, *<Data>*, is used for illustration purposes only. In an actual DiffGram, the *<Data> </Data>* block represents a DataSet or a row of a DataTable. Instead of the *<Data> </Data>* block, the DiffGram format contains the current data, whether it has been modified or not. An element, or row, that has been modified is identified with the *diffgr:hasChanges* annotation.

<diffgr:before>: This block of the DiffGram format contains the original version of a row. Elements in this block are matched to elements in the *<Data> </Data>* block using the *diffgr:id* annotation.

<diffgr:errors>: This block of the DiffGram format contains error information for a particular row in the *<Data> </Data>* block. Elements in this block are matched to elements in the *<Data> </Data>* block using the *diffgr:id* annotation.

The DiffGram also uses the following annotation that is defined in the DiffGram namespace *urn:schemas-microsoft-com:xml-diffgram-v1*:

id: Used to match the elements in the *<diffgr:before>* and *<diffgr:errors>* blocks to elements in the *<Data>* block. Values with the *diffgr:id* annotation are in the form *[TableName][RowIdentifier]* (for example: *<products diffgr:id="Products1">*).

parentId: Identifies which element from the *<Data>* block is the parent element of the current element. Values with the *diffgr:parentId* annotation are in the

form [TableName][RowIdentifier] (for example: `<Orders diffgr:parentId="Products1">`).

hasChanges: Identifies a row in the `<Data>` block as modified. The `hasChanges` annotation can have one of the following three values:

inserted: Identifies an added row

modified: Identifies a modified row that contains an original row version in the `<diffgr:before>` block. Deleted rows will have an original row version in the `<diffgr:before>` block, but there will be no annotated element in the `<Data>` block.

descent: Identifies an element where one or more children from a parent-child relationship have been modified

hasErrors: Identifies a row in the `<Data>` block with a `RowError`. The error element is placed in the `<diffgr:errors>` block.

Error: Contains the error description text of the `RowError` for a particular element in the `<diffgr:errors>` block

There are also two other annotations that are used by `DataSet` when reading and writing contents as a `DiffGram`. They are defined in the namespace `urn:schemas-microsoft-com:xml-msdata`.

RowOrder: `DataSet` to preserve the row order of the original data and identify the index of a row in a particular `DataTable`

Hidden: Identifies a column as having a `ColumnMapping` property set to `MappingType.Hidden`. The attribute is written in the format `msdata:hidden[ColumnName]="value"`.

For example:

```
<Products diffgr:id="Products1"
  msdata:hiddenSupplierTitle="Primary">.
```

Note that hidden columns are only written as a DiffGram attribute if they contain data. Otherwise, they are ignored.

Let's look at a sample DiffGram. Below is the start or header section:

```
<diffgr:diffgram
  xmlns:msdata="urn:schemas-microsoft-
    com:xml-msdata"
  xmlns:diffgr="urn:schemas-microsoft-
    com:xml-diffgram-v1">
```

Below is the data section:

```
<ProductsDataSet>

  <Products diffgr:id="Products1" msdata:rowOrder="0"
    diffgr:hasChanges="modified">
    <ProductID>SMCHEESE001</CustomerID>
    <productname>Smelly Cheese</productname>
    <price format="dollar">100.99</price>
    <expirydate>01/01/2009</expirydate>

  </Products>

  <Products diffgr:id="Products2" msdata:rowOrder="1"
    diffgram:hasErrors="true">
    <ProductID>SFCHEESE001</CustomerID>
    <productname>Soft Cheese</productname>
    <price format="dollar">79.99</price>
    <expirydate>01/01/2003</expirydate>

  </Products>

  <Products diffgr:id="Products3" msdata:rowOrder="2">
    <ProductID>HLCHEESE001</CustomerID>
    <productname>Holee Cheese</productname>
    <price format="dollar">49.99</price>
    <expirydate>01/12/2003</expirydate>
```

```
</Products>  
</ProductsDataSet>
```

Below is the Diffgr section:

```
<diffgr:before>  
  <Products diffgr:id="Products1" msdata:rowOrder="0">  
    <ProductID>SMCHEESE001</CustomerID>  
    <productname>Smelly Cheese</productname>  
    <price format="dollar">150.99</price>  
    <expirydate>01/01/2009</expirydate>  
  
  </Products>  
  
</diffgr:before>  
  
<diffgr:errors>  
  <Products diffgr:id="Products2" diffgr:  
    Error="Optimistic concurrency violation.">  
  
  </Products >  
  
</diffgr:errors>  
  
</diffgr:diffgram>
```

In the sample DiffGram above, you should note that the price for Smelly Cheese has changed from \$150.99 to \$100.99. This is probably because this product does not sell very well. Also, Soft Cheese caused an error with the message “Optimistic concurrency violation.” This could probably be due to the fact that the record for Soft Cheese has changed since it was last retrieved from the database.

Working with ReadXml

With ADO .NET, you can create a DataSet from an XML source and an XML document from a DataSet. You have great flexibility with how the data is read and the XML document is created.

The ReadXml method of the DataSet object is used to fill it with data from an XML data source.

The following table describes the ReadXml method:

Method	Description
ReadXml	Filling the DataSet with data from an XML data source takes two arguments: the XML data source and an optional XmlReadMode. The data source can be a file, a stream, or an XmlReader object. ReadXml also creates the relational schema of the DataSet depending on the specified XmlReadMode and whether or not a relational schema already exists.

The following table describes the values that XmlReadMode can have:

XmlReadMode	Description
Auto	This is the default value. ReadXml examines the XML source and chooses the most appropriate option according to the following rules: If XML source is a DiffGram, DiffGram is used. If the DataSet contains a schema or the XML source contains an inline schema, ReadSchema is used. Otherwise, InferSchema is used. If you know the format of the XML source, for optimal performance it is best that you explicitly specify the XmlReadMode instead of using auto.
ReadSchema	Reads any inline schema and loads the data and schema in the DataSet. If the DataSet already contains a schema, new tables are added to the existing schema from the inline schema. If any tables in the inline schema already exist in the DataSet, an exception is thrown.

XmlReadMode	Description
ReadSchema (cont.)	ReadSchema does not allow you to modify existing tables in the schema. Inline schema is defined using XML Schema Definition (XSD) language.
IgnoreSchema	Loads data onto the DataSet using the existing schema and ignoring any inline schema. Any data that does not conform to the schema is discarded. If the DataSet does not contain any schema, no data is loaded. If the data is a DiffGram, IgnoreSchema has the same functionality as DiffGram.
InferSchema	Ignores any inline schema, infers the schema from the structure of the XML data, and then loads the data. If the DataSet contains a schema, it is extended by adding new tables where there is no existing table or adding columns to existing tables. If an inferred table already exists with a different namespace or if any inferred columns conflict with existing columns, ReadXml throws an exception.
DiffGram	Reads a DiffGram and adds the data to the current schema. It merges new rows with existing rows using the unique identifier values to match rows.
Fragment	Reads multiple XML fragments until the end of the stream is reached. Fragments that match the DataSet schema are appended to their respective table. Any other fragments are discarded.

► **Note:** If you use an XmlReader object as the data source and the XmlReader is positioned part of the way into an XML document, ReadXml will read the next element and treat it as the root element. ReadXml will continue reading up to and until the end of the element node only. This does not apply if you are using the fragment XmlReadMode.

ReadXml processes DTD schema differently. If the data source contains entries defined in a DTD schema and the data source is specified as a filename, a stream, or a non-validating XmlReader, ReadXml throws an exception. For XML data with DTD entries, you must first create an XmlValidatingReader object with the EntityHandling property set to ExpandEntities, and then use it as the data source to ReadXml. XmlValidatingReader will expand the entities before it is read by the DataSet.

Writing XML from DataSet

We have seen how XML documents can be used as a data source for a DataSet. The DataSet can also generate XML data from the data and schema it contains. The XML can be generated with or without its schema. If you included schema information inline, then XML Schema Definition (XSD) language is used. The schema contains table definitions, relations, and constraints.

The following table describes the methods that you can use to write XML:

Method	Description
GetXml	Returns the XML in a string and does not take any arguments. Only the data is returned. To get the schema, you must use the GetXmlSchema method.
GetXmlSchema	Returns the XML schema in a string and does not take any arguments
WriteXml	Writes the XML document on the specified data target, which takes two arguments. The first is the data target, and the second is an optional XmlWriteMode. The data target can be a stream, a file, or an XmlWriter object.

As with ReadXml, WriteXml allows you great flexibility in how the XML document is created in the data target. When XML data is written from a DataSet, only the current version of the rows are written. You can, if you wish, specify that the data should be written as a DiffGram. This way, both original and current values for the rows are included.

The following table describes the values that XmlWriteMode can have:

XmlWriteMode	Description
IgnoreSchema	This is the default value. The DataSet writes the XML data without an XML schema.
WriteSchema	Writes the current contents of the DataSet as XML with the relational structure as inline XML Schema
DiffGram	Writes the XML as a DiffGram, including original and current values of rows

You can also specify how a column of a table is written in XML by changing the ColumnMapping property of the DataColumn object. The following table shows the different MappingType values that the ColumnMapping property can have and the effect on the output XML.

MappingType	Effects on Output XML Data
Element	This is the default property value. The column is written as an XML element. The ColumnName is the name of the element, and the value of the current row for the column is written as the text, e.g., <ColumnName>Column Contents row</ColumnName>.

MappingType	Effects on Output XML Data
Attribute	The column is written as an XML attribute for the current row. The ColumnName is the name of the attribute, and the contents of the column are written as the value.
SimpleContent	The column contents are written as text in the XML element for the current row. You cannot use SimpleContent for columns of a table that have other columns set as element or have nested relations.
Hidden	The column is ignored and not written to the XML output.

The ColumnMapping property is also used when writing the XML Schema of the DataSet.

XML Schemas from DataSet

The schema of a DataSet can be defined, just like its tables, columns, relations, and constraints. You can write this schema in XML Schema Definition (XSD) language. The XML Schema can be generated and transported with the data in an XML document, or it can be generated separately to a data target. A data target can be a file, a data stream, a string, or an XmlWriter object. The ColumnMapping property can be used, just as with XML, to specify how a table is represented in the XML Schema.

To write the schema, the WriteXmlSchema and GetXmlSchema methods of the DataSet are used. GetXmlSchema simply returns the schema in a string. It does not take any arguments. The WriteXmlSchema method requires one argument specifying the data target.

```
...
' Declare System.IO.StreamWriter
Dim xmlSW As System.IO.StreamWriter = _
    New System.IO.StreamWriter("ProductsWriter.xsd")

' Write DataSet XML Schema to a file
ProductDS.WriteXmlSchema("ProductsFile.xsd")

' Write DataSet XML Schema to a StreamWriter
ProductDS.WriteXmlSchema(xmlSW)

' Close StreamWriter
xmlSW.Close()

' Write XML schema in a string
Dim XSDStringSchema As String = ProductDS.GetXmlSchema()
...
```

The generated XML Schema is also useful for generating typed DataSets. For more information about typed DataSets, please refer to the next section. The DataSet, along with late-bound access to values through weakly typed variables, provides access to data through strongly typed metaphors. This allows tables and columns, which form part of the DataSet, to be accessed using user-friendly names and strongly typed variables.

Typed DataSets from XSD Schema

A typed DataSet is a class that is inherited from a DataSet. As well as inheriting all the methods, events, and properties of a DataSet, a typed DataSet also provides strongly typed methods, events, and properties in relation to the schema. This means that you can access columns by name instead of using collection-based methods. Using typed DataSets has the following advantages:

- Development of a typed DataSet
- Provides for the *localization* of code

- Allows you to extend the power of the DataSet by overriding its methods or giving them more polymorphic behaviors
- Enhances the readability of code
- Makes use of Visual Studio .NET IntelliSense features (automatically completes codes as you type)
- Catches type mismatch errors at compile time rather than at run time

From an XSD XML Schema, you can generate a strongly typed DataSet using the `xsd.exe` tool provided with the .NET Framework SDK.

The syntax for using `xsd.exe` is shown here:

```
xsd.exe /d /l:C# XSDSchemaFileName.xsd  
/n:XSDSchema.Namespace
```

`/d` instructs `xsd.exe` to generate a DataSet, and `/l:` defines what language to use (C#, in this case). `/n:` is optional and instructs `xsd.exe` to also generate a namespace (in this case, `XSDSchema.Namespace`). The source file for the schema is defined as `XSDSchemaFileName.xsd`, and the output code will be in a file called `XSDSchemaFileName.cs`. Note that the extension will depend on the language used. The generated code can also be compiled as a module or library and used in an ADO .NET application.

DataSet and XmlDataDocument

The DataSet provides you with a relational view and access to data. The XML classes available in the .NET Framework provide you with a hierarchical view and access to data. Historically, the two models have been used separately. In the .NET Framework, you can have real-time synchronous access to both. The relational model represented by DataSet objects can be

synchronized with the hierarchical model represented by `XmlDataDocument` objects.

Once a `DataSet` is set to synchronize with an `XmlDataDocument`, both objects, in effect, are sharing a single set of data. Changes made through one object are automatically reflected real-time in the other object. The ability to share data between the `DataSet` and the `XmlDataDocument` gives great flexibility by allowing access to services built around `DataSet` (such as Web Forms and Windows Forms controls) and XML services (such as XML Schema Definition (XSD) language, Extensible Stylesheet Language (XSL), XSL Transformations (XSLT), and XML Path Language (XPath)) all with one set of data.

Synchronizing DataSet with XmlDataDocument

You have different options from which to choose to synchronize a `DataSet` with an `XmlDataDocument`.

The first option you have is to populate a `DataSet` with relational data and schema and then synchronize it with a new `XmlDataDocument`. This option provides a hierarchical view to existing relational data sources.

```
' Create a DataSet
Dim aDataSet As DataSet = New DataSet

'*-----*
'* Add code here to populate DataSet with schema and data*
'*-----*

' Create XmlDataDocument and link it to a DataSet
Dim xmlDataDoc As XmlDataDocument = New
    XmlDataDocument(aDataSet)
```

The other option is to populate a `DataSet` with XML Schema (which can generate a strongly typed `DataSet`), synchronize it with an `XmlDataDocument`, and then load the `XmlDataDocument` from an XML data source, such

as an XML document. This provides the reverse option, which is a relational view to existing hierarchical data sources. The name of the tables and columns in the DataSet schema must match the names of the XML elements that you want to synchronize with. The matching is case-sensitive, and non-matching elements are ignored. This allows you to have a relatively small relational window view of part of a large XML document. While XmlDocument will preserve the whole XML document, only a portion of it (the portion you need) will be exposed through the DataSet.

```
' Declare and create the DataSet
Dim aDataSet As DataSet = New DataSet

'*-----*
'* Add code here to populate the DataSet with schema only*
'*-----*

' Create XmlDocument and link it to a DataSet
Dim xmlDataDoc As XmlDocument = New
    XmlDocument(myDataSet)

' Load the XmlDocument with XML data
xmlDataDoc.Load("anXMLDocument.xml")
```

► **Note:** You cannot load an XmlDocument if it is synchronized with a DataSet that contains data. In such a case, an exception will be thrown.

So far, the options have been DataSet-centric; that is, you start from the DataSet first. A third option is XmlDocument-centric. You create a new XmlDocument and load it from an XML data source. You then access the relational view of the data using the DataSet property of the XmlDocument. As with the other options, the schema of the DataSet is important. The table and column names in the schema must match the names, in a case-sensitive manner, of the XML elements with which you want to synchronize.

```
' Create XmlDataDocument
Dim xmlDataDoc As XmlDataDocument = New XmlDataDocument

' Create the DataSet and link it to the XmlDataDocument
Dim aDataSet As DataSet = xmlDataDoc.DataSet

'*-----*
'* Add code here to populate the DataSet with schema *
'*-----*

' Load the XmlDataDocument with XML data
xmlDataDoc.Load("anXMLDocument.xml")
```

If the DataSet is populated from an XML data source using ReadXml, the returned XML using WriteXml may differ considerably from the original. The main reason is that DataSet does not maintain formatting, such as white spaces, or hierarchical information (remember that DataSet stores relational information), such as element order. The DataSet will also not contain elements that do not conform to the schema in the DataSet. This situation may not be a problem if you are only interested in the data, but if fidelity with the original XML document is required, then synchronization will maintain fidelity with the XML data source. Hierarchical element structure of the XML data source is maintained by XmlDataDocument, while at the same time allowing the required part of the data to be accessed through DataSet.

Results of synchronization between a DataSet and an XmlDataDocument may differ, depending on whether or not the DataRelation objects are nested.

Nested DataRelations

In ADO .NET DataSet, relationships between tables are maintained and represented by the DataRelation. The parent-child relationships of columns are managed solely through relations. The tables and columns, as far as the DataRelation is concerned, are separate entities. In the hierarchical XML representation of data, the parent-child

relationships are represented by nested child elements within parent elements. To facilitate nested relationships when a DataSet is synchronized with an XmlDocument, or even when XML is written through WriteXml, the DataRelation has a nested Boolean property with a default value of False. Setting the Nested property to True causes the child rows to be nested within the parent column when outputting XML data or synchronizing with XmlDocument.

Consider the following figure and the tables' relation in the Northwind sample database included with MS SQL 2000:

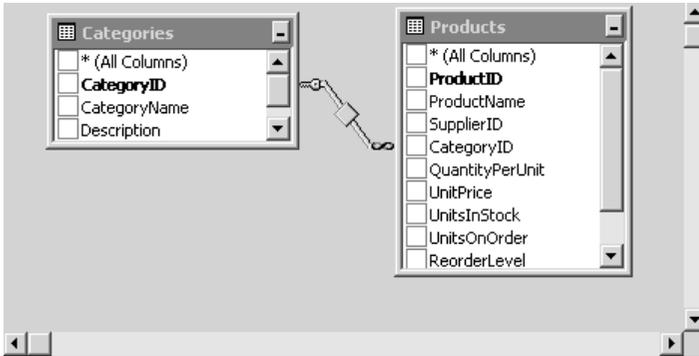


Figure 5-2: Relation between the Categories and Products tables

Now consider the following console application:

```
Imports System
Imports System.Xml
Imports System.Data
Imports System.Data.SqlClient

Module Module1

    Sub Main()

        ' Establish connection to the database
        Dim nwindConn As SqlConnection = _
            New SqlConnection("Data Source=localhost;" & _
                "Integrated Security=SSPI;Initial _
```

```

        Catalog=Northwind;")

        'Define a data adapter for category
        Dim categoryDA As SqlDataAdapter = _
            New SqlDataAdapter("SELECT CategoryID, _
                CategoryName FROM Categories", nwindConn)

        Dim productDA As SqlDataAdapter = _
            New SqlDataAdapter("SELECT ProductID, _
                CategoryID," & " ProductName, UnitPrice _
                FROM Products", nwindConn)

        'Open Database connection
        nwindConn.Open()

        'Create data set to hold data
        Dim catDS As DataSet = New DataSet
            ("CategoryProducts")

        'Populate DataSet with data from the two tables
        categoryDA.Fill(catDS, "Categories")
        productDA.Fill(catDS, "Products")

        'Close connection
        nwindConn.Close()

        'Create the relationship between the two tables
        Dim catProdcutRel As DataRelation = _
            catDS.Relations.Add("catProdcut", _
                catDS.Tables("Categories").Columns _
                ("CategoryID"), catDS.Tables _
                ("Products").Columns("CategoryID"))

        'Write data as XML
        catDS.WriteXml("NotNestedRelation.XML")

        'Set Nested to true
        catProdcutRel.Nested = True

        'Write data as XML
        catDS.WriteXml("NestedRelation.XML")

    End Sub

End Module

```

Since the default for the Nested property of the Data-Relation object is False, the child elements are not nested within the parent elements. The following code is generated in the NotNestedRelation.XML file.

```
<?xml version="1.0" standalone="yes"?>
<CategoryProducts>

...

  <Categories>
    <CategoryID>4</CategoryID>
    <CategoryName>Dairy Products</CategoryName>
  </Categories>
  <Categories>
    <CategoryID>5</CategoryID>
    <CategoryName>Grains/Cereals</CategoryName>
  </Categories>
  <Categories>
    <CategoryID>6</CategoryID>
    <CategoryName>Meat/Poultry</CategoryName>
  </Categories>

...

  <Products>
    <ProductID>2</ProductID>
    <CategoryID>1</CategoryID>
    <ProductName>Chang</ProductName>
    <UnitPrice>19</UnitPrice>
  </Products>
  <Products>
    <ProductID>3</ProductID>
    <CategoryID>2</CategoryID>
    <ProductName>Aniseed Syrup</ProductName>
    <UnitPrice>10</UnitPrice>
  </Products>
  <Products>
    <ProductID>4</ProductID>
    <CategoryID>2</CategoryID>
    <ProductName>Chef Anton's Cajun
      Seasoning</ProductName>
    <UnitPrice>22</UnitPrice>
  </Products>
```

...

`</CategoryProducts>`

Both elements, Categories and Products, in the Not-NestedRelation.XML file are siblings. However, in the NestedRelation.XML file, the Nested property of the DataRelation object is set to True. The XML output in the NestedRelation.XML file is different than that output in NotNestedRelation.XML file. The following code is generated in the NestedRelation.XML file:

```
<?xml version="1.0" standalone="yes"?>
<CategoryProducts>
  <Categories>
    <CategoryID>1</CategoryID>
    <CategoryName>Beverages</CategoryName>
    <Products>
      <ProductID>1</ProductID>
      <CategoryID>1</CategoryID>
      <ProductName>Chai</ProductName>
      <UnitPrice>18</UnitPrice>
    </Products>
    <Products>
      <ProductID>2</ProductID>
      <CategoryID>1</CategoryID>
      <ProductName>Chang</ProductName>
      <UnitPrice>19</UnitPrice>
    </Products>
  </Categories>
  ...
</CategoryProducts>

<Categories>
  <CategoryID>2</CategoryID>
  <CategoryName>Condiments</CategoryName>
  <Products>
    <ProductID>3</ProductID>
    <CategoryID>2</CategoryID>
    <ProductName>Aniseed Syrup</ProductName>
    <UnitPrice>10</UnitPrice>
  </Products>
</Categories>
```

```
<ProductID>4</ProductID>
<CategoryID>2</CategoryID>
<ProductName>Chef Anton's Cajun
    Seasoning</ProductName>
<UnitPrice>22</UnitPrice>
</Products>

...

</Categories>

...

</CategoryProducts>
```

As you can see, Products is now a child element of Categories.

Creating DataSet Relational Schema from XML Schema

For the most part, DataSet works mainly from the point of view that the source of the data is relational, and from this, we generate the XML data and schema. However, the DataSet can also do the reverse; that is, generate the relational schema from XML Schema (XSD) or even infer the schema from the XML data.

Creating from XML Schema (XSD)

In general, a table is generated for each complexType child element nested in a complexType element in the XSD schema. The structure of the table depends on the definition of the complex type. The parent complexType element usually defines the DataSet itself. If the complexType element is further nested inside another complexType element, the nested complexType element will also generate a table and will be mapped to a DataTable object within the DataSet. This usually happens when there are nested relations.

Consider the following XSD schema:

```

<?xml version="1.0" standalone="yes"?>
<xs:schema id="CategoryProducts"
  xmlns=""
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">

  <xs:element name="CategoryProducts"
    msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">

        <xs:element name="Categories">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CategoryID"
                type="xs:int"
                minOccurs="0" />

              <xs:element name="CategoryName"
                type="xs:string"
                minOccurs="0" />

              <xs:element name="Products"
                minOccurs="0"
                maxOccurs="unbounded">

                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="ProductID"
                      type="xs:int"
                      minOccurs="0" />
                    <xs:element name="CategoryID"
                      type="xs:int"
                      minOccurs="0" />
                    <xs:element name="ProductName"
                      type="xs:string"
                      minOccurs="0" />
                    <xs:element name="UnitPrice"
                      type="xs:decimal"
                      minOccurs="0" />

                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:choice>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:schema>

```

```

        </xs:element>

        </xs:sequence>
    </xs:complexType>

    </xs:element>
</xs:choice>
</xs:complexType>

<xs:unique name="Constraint1">
    <xs:selector xpath="."/></Categories" />
    <xs:field xpath="CategoryID" />
</xs:unique>

<xs:keyref name="catProdcut"
    refer="Constraint1"
    msdata:IsNested="true">
    <xs:selector xpath="."/></Products" />
    <xs:field xpath="CategoryID" />
</xs:keyref>
</xs:element>
</xs:schema>

```

With this schema, two tables will be created:

```

Categories(CategoryID, CategoryName)
Products(ProductID, CategoryID, ProductName, UnitPrice)

```

The type of each column is converted to the appropriate .NET data type. The XSD schema also causes constraints and relations to be created.

Mapping XSD Constraints to DataSet Constraints

Constraints are used to specify restrictions on elements and the values they can hold in any instance of the document. For example, if you specify the key constraint on CategoryID child element of the Categories element in the schema, the values of CategoryID in any document instance must be unique and cannot be Null. With XSD schemas, constraints are specified with elements and attributes. The common constraints used are:

- Uniqueness defined by the unique element
- A key specified by the key element
- A reference key specified by the keyref element

With the unique element, you can also specify msdata attributes:

- `msdata:ConstraintName`, where the value is used as the constraint name. Otherwise, the name attribute provides the value of the constraint name.
- `msdata:PrimaryKey`: If the value is true, the constraint is created in the DataSet with the `IsPrimaryKey` property set to True.

```
...
  <xs:unique name="Constraint1"
    msdata:ConstraintName="UCatID"
    msdata:PrimaryKey="true">

    <xs:selector xpath=" ../Categories" />
    <xs:field xpath="CategoryID" />
  </xs:unique>
...
```

The mapping process creates a unique constraint on the `CategoryID` column, as shown in the following DataSet:

```
DataSetName: CategoryProducts
TableName: Categories
  ColumnName: CategoryID
    AllowDBNull: False
    Unique: True
  ConstraintName: UCatID
    Type: UniqueConstraint
    Table: Categories
    Columns: CategoryID
    IsPrimaryKey: True
```

You can, however, have unique elements where `msdata:PrimaryKey="false."` In such a case, `AllowDBNull` will be True, meaning the column value needs to be unique or Null.

With the key element, you can also specify the same msdata attributes as with the unique element. The main difference is that with the key element, the AllowDBNull property in the DataSet constraint is always set to False. The IsPrimaryKey property depends on the value of the msdata:PrimaryKey attribute.

► **Note:** Compound keys can be specified by adding another `<xs:field/>` element for each additional column in the key.

With the keyref element, you establish relationships between elements in the document analogous to foreign keys in the relational data model. When mapping occurs, a foreign key is generated in the corresponding table in the DataSet and, by default, a relation, with the ParentTable, ChildTable, ParentColumn, and ChildColumn properties specified, is generated. The keyref element does not have the msdata:PrimaryKey attribute but can have additional msdata attributes:

- `msdata:ConstraintOnly`: If the value is True, only a constraint is created in the DataSet; otherwise, both constraint and relation are created.
- `msdata:UpdateRule`: Sets the UpdateRule constraint property to this value if specified; otherwise, the UpdateRule property is set to “Cascade.”
- `msdata>DeleteRule`: Sets the DeleteRule constraint property to this value if specified; otherwise, the DeleteRule property is set to “Cascade.”
- `msdata:AcceptRejectRule`: Sets the AcceptRejectRule constraint property to this value if specified; otherwise, the AcceptRejectRule property is set to “None.”
- `msdata:IsNested`: Sets the IsNested relation property to this value if specified; otherwise, the IsNested property is set to “False.”

```

...
  <xs:keyref name="catProdcut"
    refer=" UCatID"
    msdata:IsNested="true">
    <xs:selector xpath="//Products" />
    <xs:field xpath="CategoryID" />
  </xs:keyref>
...

```

The above will yield the following foreign key on the Products table:

```

ConstraintName: catProdcut
  Type: ForeignKeyConstraint
Table: Products
  Columns: CategoryID
RelatedTable: Categories
  RelatedColumns: CategoryID

```

DataSet Relations from XSD

There are three ways that DataSet relationships are specified in XSD:

- By inference in nested complex types
- Through `msdata:Relationship` annotation
- Through an `xs:keyref` element without the `msdata:ConstraintOnly` attribute or with the `msdata:ConstraintOnly` value set to `False`

Nested complex types indicate a parent-child relationship, which was the case between Categories and Products in our previous example. We also have already seen `keyref` elements, so we will move on to `msdata:Relationship` in the annotation element.

```

...
<xs:annotation>
  <xs:appinfo>
    <msdata:Relationship name="CustomerProductRelation"
      msdata:parent="Categories"
      msdata:child="Products"
      msdata:parentkey=" CategoryID"

```

```
msdata:childkey=" CategoryID"/>
</xs:appinfo>
</xs:annotation>
...
```

The above produces about the same result as the keyref example. However, the values of the constraint are not set. These are values, such as `IsPrimaryKey` and `AllowDBNull`, that you need to set using the constraint elements.

Inferring from XML

In certain situations, you might end up with an XML document that does not contain any inline schema nor is it one provided in a separate file. The `DataSet` can, however, still generate a schema from the XML document by analyzing the basic structure of the XML and inferring the schema.

The first step in the inference process is to determine the tables. ADO .NET first determines which elements represent tables. The remaining XML columns and relations are inferred. This is done by the following inference rules:

- Elements that have attributes are inferred as tables.
- Elements that have child elements are inferred as tables.
- Elements that repeat are inferred as a single table.
- Root elements with no attributes and no child elements inferred as columns are inferred as a `DataSet`. Otherwise, the root element is inferred as a table.
- Attributes are inferred as columns.
- Non-repeating elements with no attributes or child elements are inferred as columns.

- Elements inferred as tables that are nested within other elements also inferred as tables cause a nested `DataRelation` to be inferred between the two tables. A primary key column named "TableName_Id" is inferred, added to both tables, and used by the `DataRelation`. A `ForeignKeyConstraint` is also inferred between the two tables using the "TableName_Id" column.
- When elements are inferred as tables that contain text but have no child elements, a new column named "TableName_Text" is inferred for the text of each of the elements.

The inference process, however, is non-deterministic. Different instances of the same XML document intended to have the same schema can generate different schemas.

Consider:

```
<RootElement>  
  <AnElement>a text value</AnElement>  
  <AnElement>another text value</AnElement>  
</RootElement>
```

This infers `DataSet: RootElement`, table: `AnElement` because `AnElement` element is repeating.

Now consider:

```
<RootElement>  
  <AnElement>a third text value</AnElement>  
</RootElement>
```

This infers `DataSet: NewDataSet`, table: `RootElement`, column: `AnElement` because `AnElement` element does not have attributes, is not repeating, and has no child elements.

Summary

XML is the next stage in the development of the web standard and languages. The hierarchical model that is represented by XML data is fundamentally different from the relational model in the DataSet. ADO .NET and .NET XML classes allow the bridging of those two models in a consistent and easy manner.

XML is also the underlying message format used for Web Services. Visual Studio .NET simplifies development by shielding the developer from the complexities of establishing communication links in XML. It is all done transparently in the background. However, you also have the power to manipulate and deal with XML directly, if that is what is required. This makes ADO .NET and the .NET Framework in general a powerful and flexible tool for dealing with any type of data, including XML.

Practical ADO .NET Programming (Part One)

In This Chapter

This chapter shows the practical use of the DataSet and DataAdapter classes that are utilized to interact with the database. It is recommended that you read Chapters 2 and 3 before reading this chapter. Through the use of a simple case study, you will learn how to work with data access components, to work with XML, and to build Web Services. This chapter will demonstrate how to retrieve data from the database to a DataSet. In Chapter 7, you will learn how to update changes from a DataSet to the database.

Chapter 7 will use the Web Service we build in this chapter and continue with the case study. In this chapter, we will only look at the Web Service aspect of the case study. We will look at the clients in Chapter 7. We will also concentrate more on the data service side of things. We will not go into the details of security and maintaining user sessions, as they have little bearing on the database interaction. Instead, we will concentrate on the functions and methods that the service will expose for manipulating data from the Northwind database.

The Case Study

The Northwind Traders sample database that is included with Microsoft SQL Server 2000 contains the sales data for a fictitious company called Northwind Traders, which imports and exports specialty foods from around the world. Northwind Traders wants to provide better service for its customers and partners by allowing them access to their order information online. Furthermore, Northwind Traders wants its employees who are working in other countries to be able to amend or create new orders online.

To meet part of the requirements of Northwind Traders, you decide to build a simple Web Service to do the following:

- Allow customers to manage orders
- Allow customers to view their orders and import them in other packages

Employee functionality requirements:

- Logon
- Search orders (filter by employee and/or customer)
- Create an order
- Fill in an order
- Fill in order details
- Modify an order
- Change orders
- Change/delete order details line entries
- Delete an order entirely (pending orders only)

Customer functionality requirements:

- Logon
- Search orders (self only)
- Export orders to file

The Web Service

To meet part of these requirements, we are going to implement a Web Service. The Web Service will allow customers to have access to the data over the Internet. The Web Service will expose two types of methods:

- **Data retrieval:** These are different methods that will return data to the clients given a certain criteria.
- **Data update:** These methods allow clients to change certain information in the database. These also include deletion and creation of records.

▶ **Note:** Web Services are based on the stateless principle. That means a Web Service does not maintain state across method calls nor does it maintain any database connections.

Designing the Web Service

Before we rush in and start programming, we need to first plan and design the Web Service. The first step is to name our Web Service. For this case study, we will call our Web Service OrderProcessingWS.

OrderProcessingWS

Now that we know what we are going to call our Web Service, we need to decide what data retrieval methods we are going to expose to the clients. This is important because we will also need to use appropriate names for our methods. One thing to remember about the method names in Web Service is that they must be unique.

► **Note:** The method name of a Web Service can be different from the name of the method that implements it in the class. This is useful when you have polymorphic methods.

Data Retrieval Methods

Class Method Name	Web Service Method Name	Description
GetOrders	GetOrders_By_Customer	Parameter: CustomerID (String) Return: Dataset Returns a summary of orders for the given customer
GetOrders	GetOrders_By_Date	Parameter: FromDate (Date), ToDate (Date) Return: Dataset Returns a summary of the different orders for a given date range
GetOrders	GetOrders_By_Customer_Date	Parameter: CustomerID(String), FromDate (Date), ToDate (Date) Return: Dataset Returns a summary of orders for a given customer within a given date range

Class Method Name	Web Service Method Name	Description
GetOrder-Details	GetOrderDetails	Parameter: OrderID (Integer) Return: Dataset Returns the line entries for a given order
Get Full-Orders	GetFullOrders	Parameter: CustomerID (String) Return: Dataset Returns detailed information about an order
Get Full-Orders	GetFullOrders_By_Customer	Parameter: CustomerID (String) Return: Dataset Returns detailed information about orders for the given customer

Implementing OrderProcessingWS

Now that we have designed the skeleton for OrderProcessingWS, we can start the next phase. We will use Visual Studio .NET for the implementation and as the development environment. You will also need to have Internet Information Server (IIS) installed and accessible to your development machine. For development purposes, it is always better to have IIS installed locally. After developing and debugging, you can always move the Web Service to a production machine.

Setting Up IIS

Before you begin, first set up a virtual directory that you will use on IIS. This is not necessary, but if you let Visual Studio .NET set up the directory for you, your files will be stored in a directory under your default web site directory. Usually, this is in `c:\inetpub\wwwroot`.

To set up your virtual directory, use the following steps:

- Open the IIS management console from administrative tools.
- Expand web sites for your local machine by right-clicking Default Web Site.
- Choose New on the context menu and select Virtual Directory..., as shown in Figure 6-1.

Follow the instructions for the wizard. For the purpose of this case study, we will use a virtual directory called `OrderProcessingWS`. You can point the virtual directory to the appropriate directory where you plan to store your source code. Make sure you have the appropriate access permission to the directory.

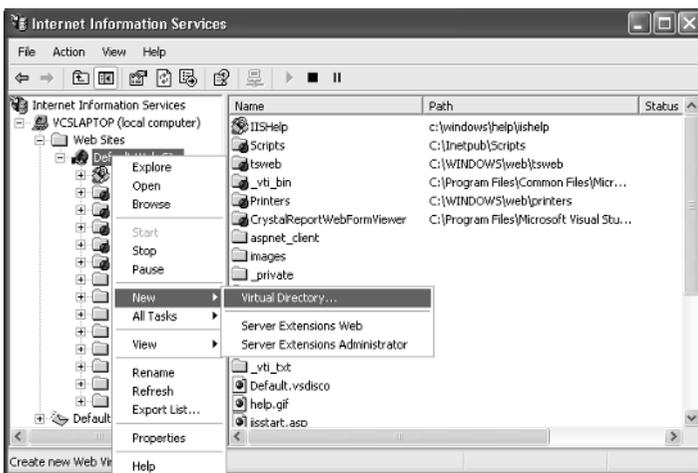


Figure 6-1: Creating a virtual directory in IIS between Categories and Products

► **Note:** ASP .NET code will not work unless the ASP .NET run time is installed. This is usually done when you install the .NET Framework. This may not be the case if IIS was not installed at the time the .NET Framework was installed. To check, ensure that you have the “Aspnet_Client” folder installed under Default Web Sites. If it is not installed, reinstall the .NET Framework.

Creating OrderProcessingWS Project

The next step is to actually create the project that you will be using for the development of OrderProcessingWS. Create a new Visual Basic ASP .NET Web Service project. Call it OrderProcessingWS. This should point to the virtual directory you created before (e.g., “http://localhost/OrderProcessingWS,” if you are using IIS locally). See Figure 6-2.

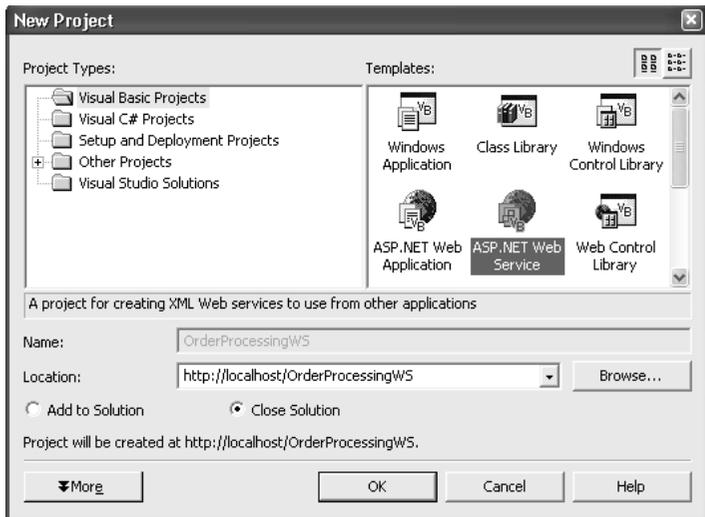


Figure 6-2: Creating the OrderProcessingWS project



Warning: Make sure you have the right level of permission on the virtual directory and you have write enabled. Otherwise, you will get an access denied error when you try to create the project.

Visual Studio creates the necessary files and directories that you will need to write the Web Service. ASP .NET application files have file extension .aspx, whereas Web Services have file extension .asmx. Visual Studio creates a default Web Service file called Service1.aspx. This file should be renamed to OrderProcessingWS.aspx. You should then set the page as the start page by right-clicking on it and choosing Set As Start Page on the context menu.

Open the OrderProcessingWS.aspx file, if it is not already open. This can be done by double-clicking the file in the Solution Explorer. This should open the file in the design view. Double-click in the design view to get to the code.



Tip: You can right-click on the file in Solution Explorer and choose View Code to access the code directly and faster.

As you can see, Visual Studio has generated some code for you. Change the class name to OrderProcessingWS. You can uncomment the suggested “Hello World” example if you wish and try it out. Delete that section of the code once you are done.

Web Service Namespace

For each Web Service you create, you will need to give it a namespace. This namespace is different from the .NET namespaces and is used to uniquely identify the Web Service on the Internet. Usually, you would use a namespace that you have control over, such as the web site of your company (for example, <http://mycompany.com/mywebservice/OrderProcessingWS/>). We will use

`http://ProgrammingADODotNET/OrderProcessingWS/` as our namespace, so go ahead and change the default `http://tempuri.org/` to `http://ProgrammingADODotNET/OrderProcessingWS/`. You should have a code section like that shown in the following figure.

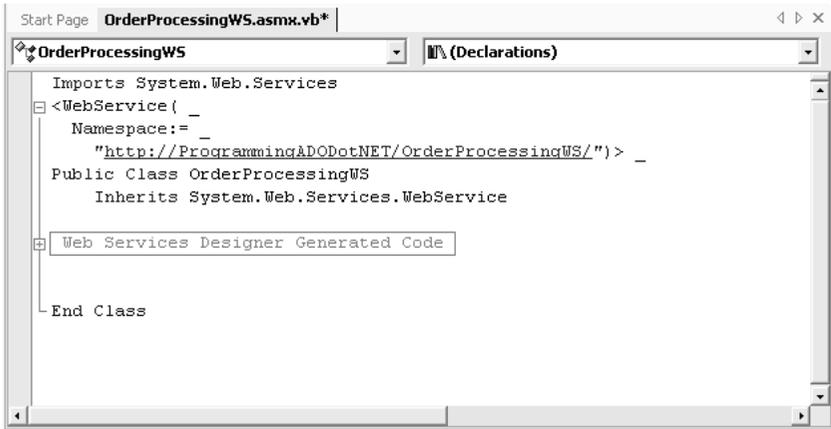


Figure 6-3: The skeleton of the class and namespace

Initialization Code

Before we add our methods, we first need to add our initialization codes. We will be using MS SQL Server, so we need to define a connection string to the server. We do not want to hard-code this data, so we will put it in the application configuration file. For Web Services, the configuration file is usually called `web.config`. Open the `web.config` file and create an `<appSettings>` section.

► **Note:** Please be aware that XML tags are case-sensitive. `<appSettings>` must be entered in exactly the same case shown here.

Then create a key called `SQLConnectionString` whose value is the connection string to your database. In the example, we are using the local database, so the data source is `127.0.0.1`. The user name, password, and

default catalog is also provided. See the code listing below as an example. As you probably have noticed, the web.config file is actually an XML file. Here is the start section of this file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="SQLConnectionString"
      value="data source=127.0.0.1;user id=sa;password=sa;
        Initial Catalog=northwind" />
  </appSettings>
  ...
```

You will now need to modify your code so that you can access the configuration file. For this, you need to use the methods of the ConfigurationSettings class. The ConfigurationSettings class is found in the System.Configuration namespace. You can reference the namespace, or you can use the imports construct so that you do not have to type the full name of the class (i.e., System.Configuration.ConfigurationSettings). We are also going to use classes in the System.Data.SqlClient namespace, so import that as well.

Expand the Web Services Designer Generated Code region in the code section of OrderProcessingWS. In the class, you need to declare a private string variable to hold the connection string. You then initialize the variable in the class constructor. The constructor is the New() method inside the class. To get the value of the SQLConnectionString key that you defined earlier in the web.config file, use the ConfigurationSettings.AppSettings method.

```
...
SQLConnectionString =
configurationSettings.AppSettings("SQLConnectionString")
...
```

See the following code sample for a detailed listing:

```
Imports System.Web.Services
Imports System.Configuration
Imports System.Data.SqlClient

<WebService( _
  Namespace:= _
  "http://ProgrammingADODotNET/OrderProcessingWS/", _
  Description:= _
  "Provides access to the order details of customers")>
Public Class OrderProcessingWS
  Inherits System.Web.Services.WebService

  #Region " Web Services Designer Generated Code "
  Private SqlConnectionString As String

  Public Sub New()
    MyBase.New()

    'This call is required by the Web Services
    'Designer.
    InitializeComponent()

    'Initialize connection string based on AppSettings
    'defined in web.config
    SqlConnectionString = _
      ConfigurationSettings.AppSettings
      ("SqlConnectionString")

  End Sub

  ...

```

To test that the application setting is working, create a method that will return the connection string. Here is an example of one:

```
<WebMethod( _
  MessageName="ConnectionString" _
)> _
Public Function ConnectionString() As String
  SqlConnectionString = SqlConnectionString
End Function

```

Once your application is complete, you can delete this function, or you can set it to private and remove the `<WebMethod>` attribute.

For easier manageability and access, create regions for the different methods that you are going to use, as shown below:

```
#Region " GetOrders Code Section "  
  
#End Region  
  
#Region " GetOrderDetails Code Section "  
  
#End Region  
  
#Region " GetFullOrders Code Section "  
  
#End Region
```

We are now ready to start coding our first method. Let's start with the `GetOrders` methods.

GetOrders Methods

Put the `GetOrders` methods in the `GetOrders Code Section` region. The source code for `GetOrders_By_Customer` is available on the companion CD.

Let's go through the different sections of the code for `GetOrders`. The first step is to declare the function. The function is declared with the `<WebMethod>` attribute in order to tell .NET to expose it as a web method of the Web Service. Because we are going to use polymorphism to declare additional `GetOrders()` methods, we must explicitly declare the name that the method will be exposed as. For this, we use the "MessageName=" property of the `<WebMethod>`:

```

<WebMethod( _
    MessageName:="GetOrders_By_Customer" _
)> _
Public Function GetOrders( _
    ByVal CustomerID As String _
) As DataSet
    '*****'
    '* Return a summary orders           *'
    '* (order header) for the           *'
    '* given customer                   *'
    '*****'

```

Notice that after the creation of the function, I have also written some header comments that describe what the function will do. This is a good habit to get into and, though tedious at first, will help you and your colleagues later when it comes to maintenance.

We then declare the DataSet, the SQL connection object, and the query string that we will use:

```

'Declare dataset to store results
Dim OrderDetailsDS As New DataSet()
OrderDetailsDS.DataSetName = "OrderDetailsDS"

'Declare SqlConnection object for connection to database
'Use the Global Private string initialized in
'MyBase.New()
Dim objConn As New SqlConnection(SQLConnectionString)

'Declare the select query string to be
'used to select the data.
'including parameters in T-SQL format and for
'optimization
'tell the database engine not to issue locks
Dim SelectQuerySTR As String = _
    "SELECT * " & _
    "FROM Orders WITH (NOLOCK)" & _
    "WHERE CustomerID = @CustomerID"

```

Since we are using DataSet to hold our data from the SQL Server, it is easier to use a SqlDataAdapter to form the bridge between the DataSet and SQL Server. Define the adapter and create an SqlCommand object in the

adapter using the query string and the connection that we declared earlier.

```
'Declare the DataAdapter that will be
'used to populate the DataSet.
Dim OrderDataAdapter As New SqlDataAdapter()

'Define the selectCommand for the DataAdapter
OrderDataAdapter.SelectCommand = _
    New SqlCommand(SelectQuerySTR, objConn)
```

In the query string, notice that we use a T-SQL parameter, @CustomerID. The SqlCommand object needs to know what value to substitute for this parameter before sending the query to the database engine. This is done by adding a parameter to the SqlCommand.

```
'Define the parameter used in the SqlCommand
'and also set its value.
OrderDataAdapter.SelectCommand.Parameters.Add _
    ("@CustomerID", CustomerID)
```

We are now ready to connect to the database and populate our DataSet using the fill method of the SqlDataAdapter. Since there is a possibility of having a connection error, we will also catch the error gracefully, if any. Once done, we clean up and close the connection. All that remains afterward is to return the DataSet.

```
'Catch possible errors
Try
    'Open the Connection to data base
    objConn.Open()

    'Populate DataSet using the DataAdapter fill method.
    OrderDataAdapter.Fill(OrderDetailsDS, "Orders")

Catch Err As Exception
    Throw New ApplicationException( _
        "Exception encountered when executing command.",
        Err)
Finally
```

```
'Close the Connection to data base
objConn.Close()
```

```
End Try
```

```
'Return the DataSet
GetOrders = OrderDetailsSDS
```

► **Note:** You do not have to worry about converting the DataSet to XML for communication over the Internet. This is done transparently by the .NET Framework.

The rest of the GetOrders methods should follow a similar pattern to the one we have just seen. The main difference is the query string and the parameters. For GetOrders_By_Date, we have:

```
'Declare the select query string to
'be used to select the data.
'including parameters in T-SQL format and for
'optimization
'tell the database engine not to issue locks
Dim SelectQuerySTR As String = _
    "SELECT * " & _
    "FROM Orders WITH (NOLOCK)" & _
    "WHERE OrderDate >= @FromDate and OrderDate <=
    @ToDate"
```

```
'(Code omitted for clarity)
```

```
'Define the parameter used in the SqlCommand
'and also set its value.
OrderDataAdapter.SelectCommand.Parameters.Add _
    ("@FromDate", FromDate)
OrderDataAdapter.SelectCommand.Parameters.Add _
    ("@ToDate", ToDate)
```

For GetOrders_By_Customer_Date, we have:

```
'Declare the select query string
'to be used to select the data.
'including parameters in T-SQL format and for
'optimization
'tell the database engine not to issue locks
```

```
Dim SelectQuerySTR As String = _
    "SELECT * " & _
    "FROM Orders WITH (NOLOCK)" & _
    "WHERE CustomerID = @CustomerID AND " & _
    "OrderDate >= @FromDate AND " & _
    "OrderDate <= @ToDate"

'(Code omitted for clarity)

'Define the parameters used in the SqlCommand
'and also set its value.
OrderDataAdapter.SelectCommand.Parameters.Add _
    ("@CustomerID", CustomerID)
OrderDataAdapter.SelectCommand.Parameters.Add _
    ("@FromDate", FromDate)
OrderDataAdapter.SelectCommand.Parameters.Add _
    ("@ToDate", ToDate)
```

The full source code for the other `GetOrders` methods is available on the companion CD.

To test the code, you can choose `Start` from the `Debug` menu or press `F5`. A web page should start listing all the available web methods for the Web Service. See Figure 6-4.

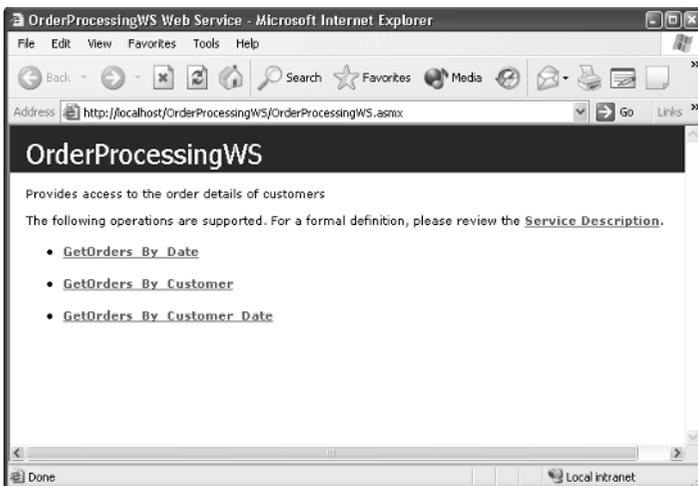


Figure 6-4: Web page generated by IIS for the `OrderProcessingWS` Web Service

Choose any of the hyperlinks to see more information about the web methods. This will take you to another page, which will allow you to test the method. See Figure 6-5.

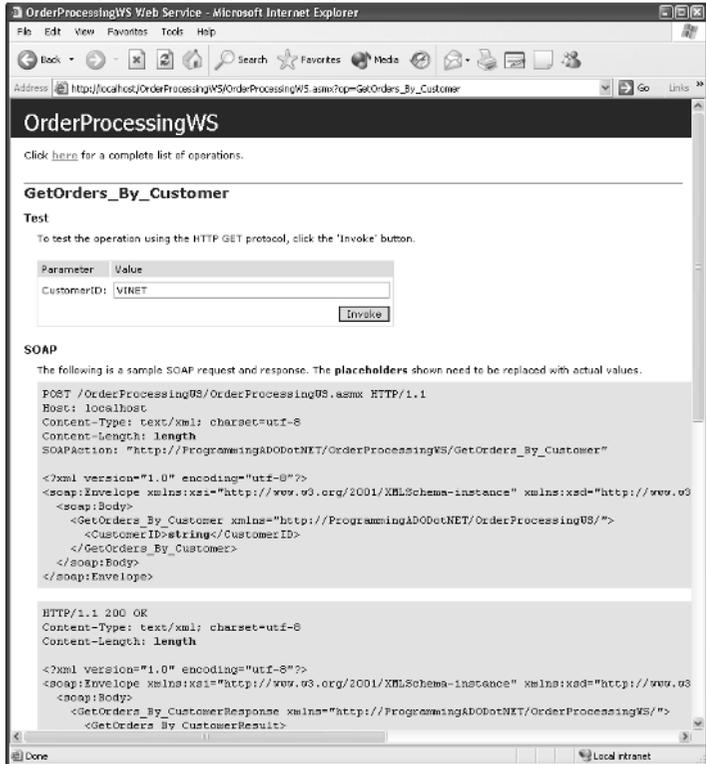


Figure 6-5: Test page for GetOrders_By_Customer

Note: You can only test methods that support HTTP get. That means, if you pass objects like DataSet as parameters you will not be able to test the method using IIS auto-generated test page. In such a case, you will have to write your own test methods.

Enter some test data in the parameter and click the Invoke button to run the test. A new page is opened that shows the result in XML format of running the method with the given data. The resulting XML is shown here:

```

<?xml version="1.0" encoding="utf-8" ?>
- <DataSet xmlns="http://ProgrammingADODotNET/
  OrderProcessingWS/">
- <xs:schema id="OrdersDS" xmlns=""
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
- <xs:element name="OrdersDS" msdata:IsDataSet="true"
  msdata:Locale="en-GB">
- <xs:complexType>
- <xs:choice maxOccurs="unbounded">
- <xs:element name="Orders">
- <xs:complexType>
- <xs:sequence>
  <xs:element name="OrderID" type="xs:int"
    minOccurs="0" />
  <xs:element name="CustomerID" type="xs:string"
    minOccurs="0" />
  <xs:element name="EmployeeID" type="xs:int"
    minOccurs="0" />
  <xs:element name="OrderDate" type="xs:dateTime"
    minOccurs="0" />
  <xs:element name="RequiredDate" type="xs:dateTime"
    minOccurs="0" />
  <xs:element name="ShippedDate" type="xs:dateTime"
    minOccurs="0" />
  <xs:element name="ShipVia" type="xs:int"
    minOccurs="0" />
  <xs:element name="Freight" type="xs:decimal"
    minOccurs="0" />
  <xs:element name="ShipName" type="xs:string"
    minOccurs="0" />
  <xs:element name="ShipAddress" type="xs:string"
    minOccurs="0" />
  <xs:element name="ShipCity" type="xs:string"
    minOccurs="0" />
  <xs:element name="ShipRegion" type="xs:string"
    minOccurs="0" />
  <xs:element name="ShipPostalCode" type="xs:string"
    minOccurs="0" />

```

```

<xs:element name="ShipCountry" type="xs:string"
  minOccurs="0" />
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:choice>
</xs:complexType>
</xs:element>
</xs:schema>
- <diffgr:diffgram xmlns:msdata=
  "urn:schemas-microsoft-com:xml-msdata"
  xmlns:diffgr="urn:schemas-microsoft-com:
  xml-diffgram-v1">
- <OrdersDS xmlns="">
- <Orders diffgr:id="Orders1" msdata:rowOrder="0">
  <OrderID>10248</OrderID>
  <CustomerID>VINET</CustomerID>
  <EmployeeID>5</EmployeeID>
  <OrderDate>1996-07-04T00:00:00.0000000+04:00</OrderDate>
  <RequiredDate>1996-08-01T00:00:00.0000000+04:00</
  RequiredDate>
  <ShippedDate>1996-07-16T00:00:00.0000000+04:00</
  ShippedDate>
  <ShipVia>3</ShipVia>
  <Freight>32.38</Freight>
  <ShipName>Vins et alcools Chevalier</ShipName>
  <ShipAddress>59 rue de l'Abbaye</ShipAddress>
  <ShipCity>Reims</ShipCity>
  <ShipPostalCode>51100</ShipPostalCode>
  <ShipCountry>France</ShipCountry>
  </Orders>
- <Orders diffgr:id="Orders2" msdata:rowOrder="1">
  <OrderID>10274</OrderID>
  <CustomerID>VINET</CustomerID>
  <EmployeeID>6</EmployeeID>
  <OrderDate>1996-08-06T00:00:00.0000000+04:00</OrderDate>
  <RequiredDate>1996-09-03T00:00:00.0000000+04:00</
  RequiredDate>
  <ShippedDate>1996-08-16T00:00:00.0000000+04:00</
  ShippedDate>
  <ShipVia>1</ShipVia>
  <Freight>6.01</Freight>
  <ShipName>Vins et alcools Chevalier</ShipName>
  <ShipAddress>59 rue de l'Abbaye</ShipAddress>
  <ShipCity>Reims</ShipCity>

```

```

    <ShipPostalCode>51100</ShipPostalCode>
    <ShipCountry>France</ShipCountry>
  </Orders>
- <Orders diffgr:id="Orders3" msdata:rowOrder="2">
  <OrderID>10295</OrderID>
  <CustomerID>VINET</CustomerID>
  <EmployeeID>2</EmployeeID>
  <OrderDate>1996-09-02T00:00:00.0000000+04:00</OrderDate>
  <RequiredDate>1996-09-30T00:00:00.0000000+04:00</
    RequiredDate>
  <ShippedDate>1996-09-10T00:00:00.0000000+04:00</
    ShippedDate>
  <ShipVia>2</ShipVia>
  <Freight>1.15</Freight>
  <ShipName>Vins et alcools Chevalier</ShipName>
  <ShipAddress>59 rue de l'Abbaye</ShipAddress>
  <ShipCity>Reims</ShipCity>
  <ShipPostalCode>51100</ShipPostalCode>
  <ShipCountry>France</ShipCountry>
  </Orders>
- <Orders diffgr:id="Orders4" msdata:rowOrder="3">
  <OrderID>10737</OrderID>
  <CustomerID>VINET</CustomerID>
  <EmployeeID>2</EmployeeID>
  <OrderDate>1997-11-11T00:00:00.0000000+04:00</OrderDate>
  <RequiredDate>1997-12-09T00:00:00.0000000+04:00</
    RequiredDate>
  <ShippedDate>1997-11-18T00:00:00.0000000+04:00</
    ShippedDate>
  <ShipVia>2</ShipVia>
  <Freight>7.79</Freight>
  <ShipName>Vins et alcools Chevalier</ShipName>
  <ShipAddress>59 rue de l'Abbaye</ShipAddress>
  <ShipCity>Reims</ShipCity>
  <ShipPostalCode>51100</ShipPostalCode>
  <ShipCountry>France</ShipCountry>
  </Orders>
- <Orders diffgr:id="Orders5" msdata:rowOrder="4">
  <OrderID>10739</OrderID>
  <CustomerID>VINET</CustomerID>
  <EmployeeID>3</EmployeeID>
  <OrderDate>1997-11-12T00:00:00.0000000+04:00</OrderDate>
  <RequiredDate>1997-12-10T00:00:00.0000000+04:00</
    RequiredDate>

```

```

<ShippedDate>1997-11-17T00:00:00.0000000+04:00</
  ShippedDate>
<ShipVia>3</ShipVia>
<Freight>11.08</Freight>
<ShipName>Vins et alcools Chevalier</ShipName>
<ShipAddress>59 rue de l'Abbaye</ShipAddress>
<ShipCity>Reims</ShipCity>
<ShipPostalCode>51100</ShipPostalCode>
<ShipCountry>France</ShipCountry>
</Orders>
</OrdersDS>
</diffgr:diffgram>
</DataSet>

```

The result shows both the XML Schema and the XML data.

GetOrderDetails Methods

The GetOrderDetails methods follow the same principle as the GetOrders methods. Again, the only difference is the query and parameter, so I will not go into detail about each section of the code.

```

#Region " GetOrderDetails Code Section "

<WebMethod(> _
Public Function GetOrderDetails( _
    ByVal OrderID As Integer _
) As DataSet
    '*****'
    '* Return the item entries          *'
    '* for the given order              *'
    '*****'

    'Declare dataset to store results
    Dim OrderEntriesDS As New DataSet()
    OrderEntriesDS.DataSetName = "OrderEntriesDS"

    'Declare SqlConnection object for connection to
    'database
    'Use the Global Private string initialized in
    'MyBase.New()

```

```
Dim objConn As New SqlConnection
    (SqlConnectionString)

'Define the select query string to be used
'to select the data.
'including parameters in T-SQL format and for
'optimization
'tell the database engine not to issue locks
Dim SelectQuerySTR As String = _
    "SELECT * " & _
    "FROM [Order Details] WITH (NOLOCK)" & _
    "WHERE OrderID = @OrderID"

'Declare the DataAdapter that will be
'used to populate the DataSet.
Dim OrderDataAdapter As New SqlDataAdapter()

'Define the selectCommand for the DataAdapter
OrderDataAdapter.SelectCommand = _
    New SqlCommand(SelectQuerySTR, objConn)

'Define the parameter used in the SqlCommand
'and also set its value.
OrderDataAdapter.SelectCommand.Parameters.Add _
    ("@OrderID", OrderID)

'Catch possible errors
Try
    'Open the Connection to database
    objConn.Open()

    'Populate the DataSet using the DataAdapter
    'fill method.
    OrderDataAdapter.Fill(OrderEntriesDS, "Orders
    Details")

Catch Err As Exception
    Throw New ApplicationException( _
        "Exception encountered when executing
        command.", Err)
Finally

    'Close the Connection to database
    objConn.Close()
```

```

End Try

'Return the DataSet
GetOrderDetails = OrderEntriesDS

End Function

#End Region

```

Note that we only have one `GetOrderDetails` method, and we want to use the same name for the web method. Therefore, we do not have to specify `MessageName:= "GetOrderDetails "` for the `<WebMethod>` attribute.

GetFullOrders Methods

So far, we have used the `DataSet` as a transporter for a single table. We have not yet used multiple tables or added relations to the `DataSet`. With `GetFullOrders`, we need to get a snapshot of the database into the `DataSet`. This means that we will also have to allow for relationships between the two tables. Figure 6-6 shows the tables, columns, and relationships that we will need in the `DataSet`.

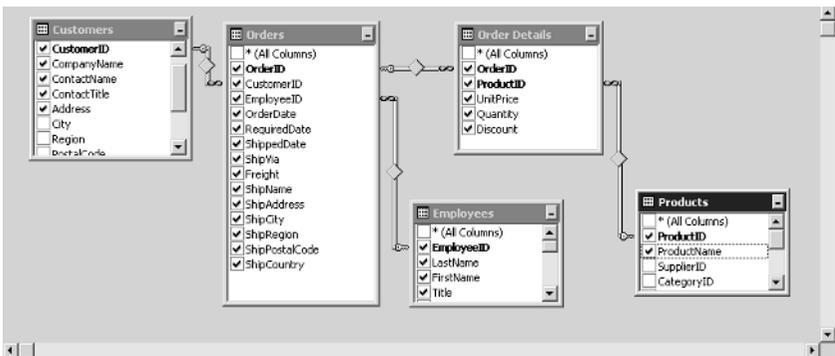


Figure 6-6: Tables, columns, and relationships for `GetFullOrders` methods

To help with the coding, it would be useful to create a typed DataSet. This will keep us from having to write code to create the proper schema. To do this, add a DataSet to the project and call it “OrdersDs.xsd.” The design should replicate what is shown in Figure 6-7.

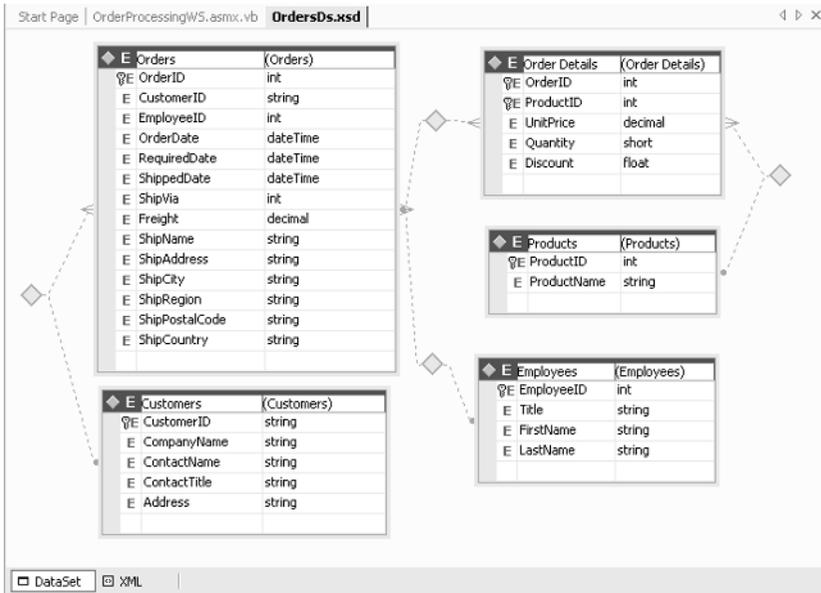


Figure 6-7: Design view of OrdersDs.xsd

The OrdersDs.xsd file is provided on the companion CD. You can use it instead of having to create your own. Once you have created the file, ensure that the Schema menu option Generate DataSet is selected; if not, select it. This will generate the DataSet for you from the XSD file, which, if you remember, is an XML Schema file. However, if you want to manually create the schema yourself, it is shown here:

```

<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="Dataset1" targetNamespace=
"http://ProgrammingADODotNET/OrderProcessingWS/
OrdersDS.xsd"
  elementFormDefault="qualified"
  attributeFormDefault="qualified"
  xmlns="http://ProgrammingADODotNET/OrderProcessingWS/
OrdersDS.xsd"
  xmlns:mstns=
"http://ProgrammingADODotNET/OrderProcessingWS/
OrdersDS.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
<xs:element name="OrdersDS" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="Orders">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="OrderID" type="xs:int"
              minOccurs="0" />
            <xs:element name="CustomerID" type="xs:string"
              minOccurs="0" />
            <xs:element name="EmployeeID" type="xs:int"
              minOccurs="0" />
            <xs:element name="OrderDate" type="xs:dateTime"
              minOccurs="0" />
            <xs:element name="RequiredDate"
              type="xs:dateTime" minOccurs="0" />
            <xs:element name="ShippedDate"
              type="xs:dateTime" minOccurs="0" />
            <xs:element name="ShipVia" type="xs:int"
              minOccurs="0" />
            <xs:element name="Freight" type="xs:decimal"
              minOccurs="0" />
            <xs:element name="ShipName" type="xs:string"
              minOccurs="0" />
            <xs:element name="ShipAddress"
              type="xs:string" minOccurs="0" />
            <xs:element name="ShipCity"
              type="xs:string" minOccurs="0" />
            <xs:element name="ShipRegion"
              type="xs:string" minOccurs="0" />
            <xs:element name="ShipPostalCode"
              type="xs:string" minOccurs="0" />
          
```

```
<xs:element name="ShipCountry"
  type="xs:string" minOccurs="0" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="Order_x0020_Details">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="OrderID" type="xs:int"
        minOccurs="0" />
      <xs:element name="ProductID" type="xs:int"
        minOccurs="0" />
      <xs:element name="UnitPrice"
        type="xs:decimal" minOccurs="0" />
      <xs:element name="Quantity" type="xs:short"
        minOccurs="0" />
      <xs:element name="Discount" type="xs:float"
        minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Customers">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="CustomerID"
        type="xs:string" minOccurs="0" />
      <xs:element name="CompanyName"
        type="xs:string" minOccurs="0" />
      <xs:element name="ContactName"
        type="xs:string" minOccurs="0" />
      <xs:element name="ContactTitle"
        type="xs:string" minOccurs="0" />
      <xs:element name="Address" type="xs:string"
        minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Employees">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="EmployeeID" type="xs:int"
        minOccurs="0" />
      <xs:element name="Title" type="xs:string"
        minOccurs="0" />
      <xs:element name="FirstName" type="xs:string"
```

```

        minOccurs="0" />
        <xs:element name="LastName" type="xs:string"
            minOccurs="0" />
    </xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="Products">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProductID" type="xs:int"
                minOccurs="0" />
            <xs:element name="ProductName"
                type="xs:string" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:choice>
</xs:complexType>
<xs:key name="OrdersPK" msdata:PrimaryKey="true">
    <xs:selector xpath="."/mstns:Orders" />
    <xs:field xpath="mstns:OrderID" />
</xs:key>
<xs:key name="EmployeesPK" msdata:PrimaryKey="true">
    <xs:selector xpath="."/mstns:Employees" />
    <xs:field xpath="mstns:EmployeeID" />
</xs:key>
<xs:key name="OrdersDetailsPK" msdata:PrimaryKey="true">
    <xs:selector xpath="."/mstns:Order_x0020_Details" />
    <xs:field xpath="mstns:OrderID" />
    <xs:field xpath="mstns:ProductID" />
</xs:key>
<xs:key name="CustomersPK" msdata:PrimaryKey="true">
    <xs:selector xpath="."/mstns:Customers" />
    <xs:field xpath="mstns:CustomerID" />
</xs:key>
<xs:key name="ProductsPK" msdata:PrimaryKey="true">
    <xs:selector xpath="."/mstns:Products" />
    <xs:field xpath="mstns:ProductID" />
</xs:key>
<xs:keyref name="ProductsOrder_x005F_x0020_Details"
    refer="ProductsPK" msdata:ConstraintOnly="true">
    <xs:selector xpath="."/mstns:Order_x0020_Details" />
    <xs:field xpath="mstns:ProductID" />
</xs:keyref>
<xs:keyref name="OrdersOrder_x005F_x0020_Details"

```

```

refer="OrdersPK" msdata:ConstraintOnly="true"
msdata>DeleteRule="Cascade"
msdata:UpdateRule="Cascade">
<xs:selector xpath="./mstns:Order_x0020_Details" />
<xs:field xpath="mstns:OrderID" />
</xs:keyref>
<xs:keyref name="CustomersOrders"
refer="CustomersPK" msdata:ConstraintOnly="true">
<xs:selector xpath="./mstns:Orders" />
<xs:field xpath="mstns:CustomerID" />
</xs:keyref>
<xs:keyref name="EmployeesOrders"
refer="EmployeesPK" msdata:ConstraintOnly="true">
<xs:selector xpath="./mstns:Orders" />
<xs:field xpath="mstns:EmployeeID" />
</xs:keyref>
</xs:element>
</xs:schema>

```

The DataSet based on OrderDS will act as a snapshot of tables and records in the database. As you can see in Figure 6-7, the tables that we need to populate in the DataSet are Customers, Employees, Products, Orders, and Order Details. OrderDS also defines a few relationships, as shown here:

Table and Columns	Relationship
Customers	Primary key is CustomerID
Employees	Primary key is EmployeeID
Orders	Primary key is OrdersID
Orders.CustomerID	Foreign key references Customers.CustomerID
Orders.EmployeeID	Foreign key references Employees.EmployeeID
Products	Primary key is ProductID
Order Details	Primary key is a compound key of OrderID and ProductID
Order Details.ProductID	Foreign key references Products.ProductID

To maintain the relationship, the DataSet must be populated in a certain order:

- The Customers and Employees tables must be populated before the Orders table.
- The Orders table must be populated before the Order Details table.
- The Products table must be populated before the Order Details table.

With the above in mind, let's go through the different sections of the GetFullOrders code.

GetFullOrders Code

First, let's look at the simplest of the GetFullOrders methods, the GetFullOrders using OrderID. First, declare the method and required variables:

```
<WebMethod(MessageName:="GetFullOrders")> _
Public Function GetFullOrders(ByVal OrderID As _
Integer)
As DataSet

    '*****'
    '* Return a snapshot of tables          *'
    '* related to the given order          *'
    '*****'

    'Declare dataset to hold tables and relations
    Dim OrderDS As New OrdersDS()

    'Declare SqlConnection object for connection to
    'database
    Dim objConn As New SqlConnection
    (SQLConnectionString)

    'Declare the DataAdapter that will be used
    'to populate the DataTables.
    Dim OrderDataAdapter As New SqlDataAdapter()
```

Notice that OrderDS is declared as type OrdersDS, which is our typed DataSet. We need to define query strings for each of the tables we are going to populate:

```
'Declare the select query string to be used
'to select the data from Orders table
'including parameters in T-SQL format and for
'optimization
'tell the database engine not to issue locks
Dim SelectOrderSTR As String = _
"SELECT OrderID, CustomerID, " & _
"EmployeeID, OrderDate, " & _
"RequiredDate, ShippedDate, " & _
"ShipVia, Freight, " & _
"ShipName, ShipAddress, " & _
"ShipCity, ShipRegion, " & _
"ShipPostalCode, ShipCountry " & _
"FROM Orders with (NOLOCK) " & _
"WHERE Orders.OrderID = @OrderID"

'Declare the select query string to be used
'for Order details
Dim SelectOrderDetailsSTR As String = _
"SELECT [Order Details].OrderID, " & _
"[Order Details].ProductID, " & _
"[Order Details].UnitPrice, " & _
"[Order Details].Quantity, " & _
"[Order Details].Discount " & _
"FROM [Order Details] WITH (NOLOCK) " & _
"WHERE [Order Details].OrderID = @OrderID"

'Declare the select query string to be used
'for Products. Note that we do not retrieve
'all products as this will make the dataset
'load unused data
Dim SelectProductsSTR As String = _
"SELECT DISTINCT Products.ProductID, " & _
"Products.ProductName " & _
"FROM [Order Details] JOIN " & _
"Products ON " & _
"[Order Details].ProductID = " & _
"Products.ProductID " & _
"WHERE [Order Details].OrderID = @OrderID"
```

```
'Declare the select query string to be used
'for Employees.
Dim SelectEmployeesSTR As String = _
"SELECT DISTINCT Employees.EmployeeID, " & _
"Employees.Title, Employees.FirstName, " & _
"Employees.LastName " & _
"FROM Orders JOIN Employees ON " & _
"Orders.EmployeeID = Employees.EmployeeID " & _
"WHERE Orders.OrderID = @OrderID"

'Declare the select query string to be used
'for Customers.
Dim SelectCustomersSTR As String = _
"SELECT DISTINCT Customers.CustomerID, " & _
"Customers.CompanyName, Customers.ContactName, " & _
"Customers.ContactTitle, Customers.Address " & _
"FROM Orders JOIN Customers ON " & _
"Orders.CustomerID = Customers.CustomerID " & _
"WHERE Orders.OrderID = @OrderID"
```

The query string makes sure that, as far as possible, only the required records are retrieved. For example, only the products found on that particular order are retrieved. This helps to reduce the size of the DataSet that will be returned to the clients. In a full-blown system, the Products table could be returned by a separate method. The client could keep a read-only copy of the list updated periodically or on demand. You will also notice that not all columns in the supporting table (Customers, Employees, and Products) are returned. Again, this is again to reduce the size of the DataSet.

Once we have defined the query string, all that is left is to fill the respective tables:

```
'Define the selectCommand for the DataAdapter
OrderDataAdapter.SelectCommand = New SqlCommand()
OrderDataAdapter.SelectCommand.Connection =
objConn

'Define the parameter used in the SqlCommand
'and also set its value.
OrderDataAdapter.SelectCommand.Parameters.Add _
```

```
        ("@OrderID", OrderID)

    Try
        'Open the Connection to database
        objConn.Open()

        OrderDataAdapter.SelectCommand.CommandText = _
            SelectProductsSTR
        'Populate the DataSet using the DataAdapter
        'fill method.
        OrderDataAdapter.Fill(OrderDS.Products)

        OrderDataAdapter.SelectCommand.CommandText = _
            SelectEmployeesSTR
        'Populate the DataSet using the DataAdapter
        'fill method.
        OrderDataAdapter.Fill(OrderDS.Employees)

        OrderDataAdapter.SelectCommand.CommandText = _
            SelectCustomersSTR
        'Populate the DataSet using the DataAdapter
        'fill method.
        OrderDataAdapter.Fill(OrderDS.Customers)

        OrderDataAdapter.SelectCommand.CommandText = _
            SelectOrderSTR
        'Populate the DataSet using the DataAdapter
        'fill method.
        OrderDataAdapter.Fill(OrderDS.Orders)

        OrderDataAdapter.SelectCommand.CommandText = _
            SelectOrderDetailsSTR
        'Populate the DataSet using the DataAdapter
        'fill method.
        OrderDataAdapter.Fill(OrderDS.Order_Details)

    Catch Err As Exception
        Throw New ApplicationException( _
            "Exception encountered when executing
            command.", Err)
    Finally

        'Close the Connection to database
        objConn.Close()
```

```

End Try

'Return the dataset
GetFullOrders = OrderDS

End Function

```

We only need to use one DataAdapter. The key to this technique is to change the CommandText property of the SqlCommand object for each table that we retrieve. If you look carefully, you will notice also that we are referencing the tables in the DataSet using the dot notation. For example, we use OrderDS.Order_Details to refer to the Order Details table. Since spaces are not allowed in the names of objects in VB .NET syntax, the space in the table name is replaced by an underscore, “_”. Therefore, “Order Details” becomes “Order_Details”. As you would expect, the parameters used in the script must also be added to the SqlCommand object. Since I have chosen the same name for that parameter in my script, I only have to define it once.

GetFullOrders_By_Customer Code

For GetFullOrders_By_Customer, the only difference is the query string and parameter:

```

'Declare the select query string to be used
'to select the data from Orders table
'including parameters in T-SQL format and for
'optimization
'tell the database engine not to issue locks
Dim SelectOrderSTR As String = _
"SELECT OrderID, CustomerID, " & _
"EmployeeID, OrderDate, " & _
"RequiredDate, ShippedDate, " & _
"ShipVia, Freight, " & _
"ShipName, ShipAddress, " & _
"ShipCity, ShipRegion, " & _
"ShipPostalCode, ShipCountry " & _

```

```

"FROM Orders with (NOLOCK) " & _
"WHERE Orders.CustomerID = @CustomerID"

'Declare the select query string to be used
'for Order details
Dim SelectOrderDetailsSTR As String = _
"SELECT [Order Details].OrderID, " & _
"[Order Details].ProductID, " & _
"[Order Details].UnitPrice, " & _
"[Order Details].Quantity, " & _
"[Order Details].Discount " & _
"FROM Orders JOIN [Order Details] " & _
"ON Orders.OrderID = [Order Details].OrderID " & _
"WHERE Orders.CustomerID = @CustomerID"

'Declare the select query string to be used
'for Products. Note that we do not retrieve
'all products as this will make the dataset
'load unused data
Dim SelectProductsSTR As String = _
"SELECT DISTINCT Products.ProductID, " & _
"Products.ProductName " & _
"FROM Orders JOIN [Order Details] " & _
"ON Orders.OrderID = [Order Details].OrderID " & _
"JOIN Products ON " & _
"[Order Details].ProductID = " & _
"Products.ProductID " & _
"WHERE Orders.CustomerID = @CustomerID"

'Declare the select query string to be used
'for Employees.
Dim SelectEmployeesSTR As String = _
"SELECT DISTINCT Employees.EmployeeID, " & _
"Employees.Title, Employees.FirstName, " & _
"Employees.LastName " & _
"FROM Orders JOIN Employees ON " & _
"Orders.EmployeeID = Employees.EmployeeID " & _
"WHERE Orders.CustomerID = @CustomerID"

'Declare the select query string to be used
'for Customers.
Dim SelectCustomersSTR As String = _
"SELECT Customers.CustomerID, " & _
"Customers.CompanyName, Customers.ContactName, " & _

```

```
"Customers.ContactTitle, Customers.Address " & _  
"FROM Customers " & _  
"WHERE Customers.CustomerID = @CustomerID"
```

The SELECT query used to filter the rows in this case is slightly more complicated because we always have to join with the Orders table to be able to get the CustomerID. This is particularly apparent in the Products table where the join is two levels deep. In this case, the Products table is joined to the Order Details table via the ProductID column. In turn, the Order Details table is joined to the Orders table via the OrderID column. Order is then restricted using the CustomerID column. Since the same products may be on different orders for the same customer, we must also use the DISTINCT qualifier in the query so that we do not get duplicate products.

Summary

In this chapter, we looked at how to retrieve data from the database to the DataSet. We went through the steps required to set up the case study, including setting up Microsoft Internet Information Server (IIS) to use Web Service. We also saw how to use and create a typed DataSet and how to retrieve data from multiple tables into one DataSet.

In Chapter 7, we will continue with the Web Service and look at the methods and techniques required to update the database via the Web Service and DataSet.

Practical ADO .NET Programming (Part Two)

In This Chapter

In Chapter 6 we concentrated on the data retrieval aspect of the Web Service. In this chapter, we will expand the Web Service to include data update methods.

We will look at how to set up the DataSet to avoid concurrency issues and how to update data via stored procedures instead of scripts.

Data Update Methods

Updating data via the DataSet can be tricky. In a previous version of ADO, the update was done row by row manually by looping through each row and running the relevant query. You can still do this with DataSet. You can reference each row of a DataTable in a DataSet, loop through each one, and run the appropriate query. However, the update method of the DataAdapter can take care of all the required looping for you.

There is still one problem with the update method. It does all updates, including deletes, inserts, and, obviously, updates, in the order that it retrieves the rows from the DataTable. It checks the RowState property and

does the required INSERT, DELETE, or UPDATE command. If you require updates to be done in a particular order, you must filter the DataSet. In the case study, we need to control the order of the updates for the Order Details and Orders tables. You need to create separate update methods to take care of the deletes, updates, and inserts for both tables.

The Update Functions

Update Methods	Description
UpdateOrderDetails	Parameter: ByRef DataTable Updates the Order Details table
InsertOrderDetails	Parameter: ByRef DataTable Inserts new rows in the Order Details table
DeleteOrderDetails	Parameter: ByRef DataTable Deletes rows in the Order Details table
UpdateOrders	Parameter: ByRef DataTable Updates the Orders table
InsertOrders	Parameter: ByRef DataTable Inserts new rows in the Orders table
DeleteOrders	Parameter: ByRef DataTable Deletes rows in the Orders table

The update functions all follow a similar pattern—they initialize variables, define the query string, define parameters, and finally execute the query string, including trapping any errors that may arise.

The Protected Order Details Update Methods

Let's look at the simpler update methods: the update methods for the Order Details table. They are simpler because the Order Details table is a child table and does not have dependent tables.

The DeleteOrderDetails Method

The DeleteOrderDetails method is the simplest of the three because it requires fewer parameters. In fact, you only need to supply parameters for the primary key columns, OrderID and ProductID. As is standard, the first step is to declare the method and the required variables.

```
Protected Function DeleteOrderDetails _
    (ByRef OrderDetailsTB As DataTable)

    'Declare SqlConnection object for connection to database
    'Use the Global Private string initialized in
    'MyBase.New()
    Dim objConn As New SqlConnection(SQLConnectionString)

    'Declare the DataAdapter that will be
    'used to populate the DataSet.
    Dim OrderDataAdapter As New SqlDataAdapter()

    OrderDataAdapter.DeleteCommand = New SqlCommand()

    'Declare the DataAdapter that will be
    'used to populate the DataSet.
    Dim workParm As SqlParameter

    'Declare the query string
    'that will perform the update
    Dim DeleteOrderDetailsSTR As String = _
        "DELETE FROM [Order Details] " & _
        " WHERE " & _
        "OrderID = @OrderID AND " & _
        "ProductID = @ProductID "
```

Notice that we pass along a DataTable instead of a DataSet as a parameter. This is possible because the method will not be exposed as a web method.

Furthermore, these update methods will only be used by other methods in the class or by descendants of the class, so we declare it as protected. The initialization also includes the query string required for the delete. Make a note of the parameters used because in the next section, we need to define these parameters.

```
'Create new parameter and reference it
'Set the source and version
workParm = OrderDataAdapter.DeleteCommand. _
    Parameters.Add("@OrderID", SqlDbType.Int)
workParm.SourceColumn = "OrderID"
workParm.SourceVersion = DataRowVersion.Original

workParm = OrderDataAdapter.DeleteCommand. _
    Parameters.Add("@ProductID", SqlDbType.Int)
workParm.SourceColumn = "ProductID"
workParm.SourceVersion = DataRowVersion.Original
```

We use the reference variable `workParm` to refer to each parameter that we create to reduce the amount of code we have to type. When a parameter is created, we also define the data type that is used in the database. In the database, both `OrderID` and `ProductID` are integers represented in the code by the enumerated data type `SqlDbType.Int`. We define the source of the parameter as a column in the `DataTable`. Next, we define which version of the row to use. Since we are going to delete the row, we need to make sure that we use the original version of the key columns so that we refer to the right row. If we use the default version, which is `DataRowVersion.Current`, we might end up deleting the wrong records. This can happen if the columns were changed and deleted in the `DataSet`.

After we have done all the required settings, we can then go ahead and do the update.

```
'Set command text and connection
OrderDataAdapter.DeleteCommand.CommandText = DeleteOrderDetailsSTR
OrderDataAdapter.DeleteCommand.Connection = objConn

Try
  'Open the Connection to database
  objConn.Open()

  'Next process updates.
  DeleteOrderDetails = OrderDataAdapter.Update(OrderDetailsTB.Select _
    (Nothing, Nothing, DataRowState.Deleted))

Catch ConcurrencyErr As DbConcurrencyException
  Throw ConcurrencyErr

Catch Err As Exception
  Throw New ApplicationException("Exception encountered when executing command.",
    Err)

Finally

  'Close the Connection to database
  objConn.Close()

End Try

End Function
```

This part of the code is more or less standard. All we do is add the query string to the `DeleteCommand.CommandText` property, set the connection object to use, and do the update. We filter rows in the `DataTable` only to pass the deleted rows using the `Select` method of the `DataTable` and the `DataViewRowState.Deleted` row state filter. This will ensure that only deleted rows are passed to the update method of the `DataAdapter`, ensuring that the update method does only deletes.

► **Note:** We catch the concurrency error separately from other errors. Concurrency errors occur if the query did not affect any row. Because of the disconnected nature of the DataSet and Web Services, it is possible that the rows in the database, in this case, have been deleted by another user after the rows were last retrieved by the current user. I will discuss concurrency in more detail later in this chapter.

The InsertOrderDetails Method

The InsertOrderDetails method follows the same pattern as DeleteOrderDetails, but this time the query is slightly more complicated.

```
Dim InsertOrderDetailsSTR As String = _
    "INSERT INTO [Order Details]" & _
    "(OrderID,ProductID,UnitPrice,Quantity,Discount) " & _
    "SELECT @OrderID,@ProductID," & _
    "@UnitPrice,@Quantity,@Discount " & _
    "FROM Orders ,Products " & _
    "where Orders.OrderID = @OrderID " & _
    "AND Products.ProductID = @ProductID " & _
    "AND @ProductID not in " & _
    "( SELECT ProductID " & _
    "from [Order Details] " & _
    "where [Order Details].OrderID = @OrderID)"
```

The SQL INSERT command is used with a SELECT to restrict the insert. The SELECT returns the list of insert parameters only if the following conditions are true:

- An order with the given OrderID exists.
- A product with the given ProductID exists.
- The given ProductID is not already part of the Order Details for the given OrderID.

These conditions are mainly for concurrency reasons, to avoid inserts of products that have already been added to the order since the order was last retrieved. It also prevents addition if the order or the product no longer

exists. As for the parameter version, we now use current versions for all parameters because we are dealing with new rows.

```
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@OrderID", SqlDbType.Int)
workParm.SourceColumn = "OrderID"
workParm.SourceVersion = DataRowVersion.Current

workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@ProductID", SqlDbType.Int)
workParm.SourceColumn = "ProductID"
workParm.SourceVersion = DataRowVersion.Current

workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@UnitPrice", SqlDbType.Money)
workParm.SourceColumn = "UnitPrice"
workParm.SourceVersion = DataRowVersion.Current

workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@Quantity", SqlDbType.SmallInt)
workParm.SourceColumn = "Quantity"
workParm.SourceVersion = DataRowVersion.Current

workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@Discount", SqlDbType.Real)
workParm.SourceColumn = "Discount"
workParm.SourceVersion = DataRowVersion.Current
```

You do not need to specify the `SourceVersion` as `DataRowVersion.Current` since this is the default value, but doing so helps in the clarity of the code and removes ambiguity.

The update is similar to that of `DeleteOrder`, except this time we filter for the inserted rows.

```
OrderDataAdapter.InsertCommand.CommandText =
    InsertOrderDetailsSTR
OrderDataAdapter.InsertCommand.Connection = objConn

Try
    'Open the Connection to database
    objConn.Open()
```

```

'Next process updates.
InsertOrderDetails = OrderDataAdapter.Update( _
    OrderDetailsTB.Select( _
        Nothing, Nothing, DataRowState.Added))

Catch ConcurrencyErr As DBConcurrencyException
    Throw ConcurrencyErr

Catch Err As Exception
    Throw New ApplicationException( _
        "Exception encountered when executing command.",
        Err)

Finally

    'Close the Connection to database
    objConn.Close()

End Try

```

To get the new rows from the DataTable, we use DataRowState.Added as the filter criteria.

The UpdateOrderDetails Method

The UpdateOrderDetails method is slightly more tricky than the DeleteOrderDetails and the InsertOrderDetails methods. First, UpdateOrderDetails is declared and variables are initialized.

```

Public Function UpdateOrderDetails _
    (ByVal OrderDetailsTB As DataTable)
    As Integer

    'Declare SqlConnection object for connection to database
    'Use the Global Private string initialized in
    'MyBase.New()
    Dim objConn As New SqlConnection(SQLConnectionString)

    'Declare the DataAdapter that will be
    'used to populate the DataSet.
    Dim OrderDataAdapter As New SqlDataAdapter()

    OrderDataAdapter.UpdateCommand = New SqlCommand()

```

```
Dim workParm As SqlParameter

Dim UpdateOrderDetailsSTR As String = _
"UPDATE [Order Details] " & _
"SET UnitPrice = @NewUnitPrice, " & _
" Quantity = @NewQuantity, " & _
" Discount = @NewDiscount " & _
" WHERE OrderID = @OrigOrderID AND" & _
" ProductID = @OrigProductID"
```

The query string is simple, but do make a note that only the non-primary key columns are updated. The primary key columns are used to qualify the update. The tricky part is the next code section dealing with the parameters.

```
workParm = OrderDataAdapter.UpdateCommand. _
Parameters.Add("@OrigOrderID", SqlDbType.Int)
workParm.SourceColumn = "OrderID"
workParm.SourceVersion = DataRowVersion.Original

workParm = OrderDataAdapter.UpdateCommand. _
Parameters.Add("@OrigProductID", SqlDbType.Int)
workParm.SourceColumn = "ProductID"
workParm.SourceVersion = DataRowVersion.Original

workParm = OrderDataAdapter.UpdateCommand. _
Parameters.Add("@NewUnitPrice", SqlDbType.Money)
workParm.SourceColumn = "UnitPrice"
workParm.SourceVersion = DataRowVersion.Current

workParm = OrderDataAdapter.UpdateCommand. _
Parameters.Add("@NewQuantity", SqlDbType.SmallInt)
workParm.SourceColumn = "Quantity"
workParm.SourceVersion = DataRowVersion.Current

workParm = OrderDataAdapter.UpdateCommand. _
Parameters.Add("@NewDiscount", SqlDbType.Real)
workParm.SourceColumn = "Discount"
workParm.SourceVersion = DataRowVersion.Current
```

For the parameter referring to the primary keys, you must use the original version of the row. This ensures that you do not update the wrong row and provides

rudimentary concurrency error checks. In effect, we have also made the primary keys read-only. As for the rest of the parameters, use the current version so that the update is applied. Afterward, all that remains is to do the update, making sure that only changed rows are passed to the update method

```
OrderDataAdapter.UpdateCommand.CommandText =
    UpdateOrderDetailsSTR
OrderDataAdapter.UpdateCommand.Connection = objConn

Try
    'Open the Connection to database
    objConn.Open()

    'Next process updates.
    UpdateOrderDetails = OrderDataAdapter.Update _
        (OrderDetailsTB.Select( _
            Nothing, Nothing, DataRowState.
                ModifiedCurrent))

Catch ConcurrencyErr As DBConcurrencyException
    Throw ConcurrencyErr

Catch Err As Exception
    Throw New ApplicationException( _
        "Exception encountered when executing command.",
        Err)

Finally

    'Close the Connection to database
    objConn.Close()

End Try
```

Concurrency Issues

In the update functions, there is no advance concurrency error check that checks if the rows have changed since they were last retrieved. This is a common problem in disconnected multiuser systems. To do this check, you may need to check every column and row in the table with the corresponding columns and rows in the DataTable. This, though, is not the most efficient design. The more common solution is to add a special column that you can use for tracking row versions in each table. This column, usually a datetime field or a timestamp, is changed each time any column in the particular row is modified. This can be done by the database engine through the use of triggers.

 **Note:** In Microsoft SQL Server 2000, the Transact-SQL timestamp data type is not the same as the timestamp data type defined in the SQL-92 standard. The SQL-92 timestamp data type is equivalent to the Transact-SQL datetime data type. Future releases of the SQL Server may modify the behavior of the timestamp data type to comply with the behavior defined in the SQL-92 standard. At that time, the current timestamp data type will be replaced with a rowversion data type.

The special concurrency check field is always retrieved with any updateable records. When doing an update or delete, the original value of the concurrency field is used against that of the database, usually in the WHERE clause. If the original does not match the database current value, the update method will cause a concurrency exception. The DataAdapter update method throws a concurrency error if the update causes zero rows to be affected.

The Protected Orders Update Methods

Thus far, we have completed the update methods that we will need to update the Order Details table. To be able to modify an order, we must also add update methods to modify the order header or, more accurately, the Orders table.

The methods follow what now is a familiar pattern for the update methods we have done so far. However, to show you the power of the DataAdapter update method, we will do the updates through stored procedures.

 **Note:** For illustration purposes, I have chosen not to update Order Details through stored procedures. However, in a real-world project, the use of stored procedures is advised over direct SQL query. This is mainly for performance and security purposes. It might also be a requirement imposed on you by the database administrator.

Before we move on with the methods, we need to define the stored procedure that the method will use. You can apply the required SQL query to generate the stored procedure using the query analyzer tool that is included with Microsoft SQL Server 2000. Make sure that you apply the stored procedure to the Northwind database.

The `sp_UpdateOrders` Stored Procedure

The first part of the script is to check if the stored procedure already exists. If it does, we drop the stored procedure. This part of the script is useful, especially when debugging and writing the scripts. It keeps us from having to manually drop the stored procedure each time we apply a new version.

```
-- =====
-- Script for sp_UpdateOrders
-- =====

/*****/
/* Check if a version of the stored      */
/* procedure already exist and if so drop it */
/*****/

IF EXISTS (SELECT name
           FROM   sysobjects
           WHERE  name = N'sp_UpdateOrders'
           AND   type = 'P')
    DROP PROCEDURE sp_UpdateOrders
GO
```

Next, we create the procedure with all the required parameters.

```
CREATE PROCEDURE sp_UpdateOrders
    @OrderID integer,
    @CustomerID nchar(5),
    @EmployeeID integer,
    @OrderDate datetime,
    @RequiredDate datetime,
    @ShippedDate datetime,
    @ShipVia integer,
    @Freight money,
    @ShipName nvarchar(40),
    @ShipAddress nvarchar(60),
    @ShipCity nvarchar(15),
    @ShipRegion nvarchar(15),
    @ShipPostalCode nvarchar(10),
    @ShipCountry nvarchar(15)
AS
BEGIN
```

Finally, we define the SQL scripts that we need to run to do the required update.

```

/*****/
/* set up SQL to use transaction to allow */
/* rollback in case of error */
/*****/

BEGIN TRANSACTION

/* Run script to update Orders table */
UPDATE Orders SET
    CustomerID = @CustomerID,
    EmployeeID = @EmployeeID,
    OrderDate = @OrderDate,
    RequiredDate = @RequiredDate,
    ShippedDate = @ShippedDate,
    ShipVia = @ShipVia,
    Freight = @Freight,
    ShipName = @ShipName,
    ShipAddress = @ShipAddress,
    ShipCity = @ShipCity,
    ShipRegion = @ShipRegion,
    ShipPostalCode = @ShipPostalCode,
    ShipCountry = @ShipCountry
WHERE Orders.OrderID = @OrderID

/* check for error*/
IF (@@ERROR <> 0)
    GOTO Error

GOTO Ok

Error:
/* Error Condition, so rollback*/
ROLLBACK TRANSACTION
RETURN -1
GOTO Finally

Ok:
/*Everything is ok, so commit*/
COMMIT TRANSACTION
RETURN 0

```

Finally:

END

GO

The SQL UPDATE script uses only basic concurrency checks. The stored procedure only checks for existence of the order but does not check if the order has been changed since it was last retrieved. It will override any changes made. This method is ideal only in a segregated multiuser system. In such a system, where updating is concerned, only one user can update certain rows. For instance, our system would allow only one connection per company to the Web Service, and the company will only be allowed to update its own order.

The sp_InsertOrders Stored Procedure

As is standard, we first do the required declaration:

```
-- =====
-- Script for sp_InsertOrders
-- =====

/*****
/* Check if a version of the stored      */
/* procedure already exists and if so drop it */
*****/
IF EXISTS (SELECT name
           FROM   sysobjects
           WHERE  name = N'sp_InsertOrders'
           AND   type = 'P')
    DROP PROCEDURE sp_InsertOrders
GO

CREATE PROCEDURE sp_InsertOrders
    @OrderID integer output,
    @CustomerID nchar(5),
    @EmployeeID integer,
    @OrderDate datetime,
    @RequiredDate datetime,
    @ShippedDate datetime,
```

```

@ShipVia integer,
@Freight money,
@ShipName nvarchar(40),
@ShipAddress nvarchar(60),
@ShipCity nvarchar(15),
@ShipRegion nvarchar(15),
@ShipPostalCode nvarchar(10),
@ShipCountry nvarchar(15)
AS
BEGIN

```

Notice that the stored procedure has an output parameter. Use this technique when some column values are auto-generated and required in subsequent updates. In our case, we will need the generated value of the OrderID column for inserting related rows in Order Details.

The next step is to define a special variable “@@Now” that we will use to generate values to help retrieve the auto-generated value of OrderID. @@Now is set to the value of the current date and time. Since datetime is accurate to more than a millisecond, the chance of duplicating the value for @@Now is remote. We then do the insert, but instead of adding all values for all columns for certain datetime columns (OrderDate, RequiredDate, and ShippedDate), we set them to the value generated in @@Now.

```

/*****
/* declare a special variable to be used as */
/* a special marker in the inserted row so */
/* that we can retrieve the auto-generated */
/* value for the OrderID column */
*****/
DECLARE @@Now datetime
SET @@Now = GETDATE()

BEGIN TRANSACTION

```

```
/******  
/* Run script to insert in the Orders table */  
/* Use the datetime columns and @@Now to add */  
/* markers to record so that we can retrieve */  
/* the auto generated OrderId */  
/******  
  
Insert INTO Orders(  
    CustomerID,  
    EmployeeID,  
    OrderDate,  
    RequiredDate,  
    ShippedDate,  
    ShipVia,  
    Freight,  
    ShipName,  
    ShipAddress,  
    ShipCity,  
    ShipRegion,  
    ShipPostalCode,  
    ShipCountry )  
VALUES (  
    @CustomerID,  
    @EmployeeID,  
    @@Now,  
    @@Now,  
    @@Now,  
    @ShipVia,  
    @Freight,  
    @ShipName,  
    @ShipAddress,  
    @ShipCity,  
    @ShipRegion,  
    @ShipPostalCode,  
    @ShipCountry )  
  
/* check for error*/  
IF (@@ERROR <> 0)  
    GOTO Error
```

Once we have checked for errors with the insert and everything is fine, the next step is to retrieve the generated OrderID value. For this, OrderDate, RequiredDate, and ShippedDate are retrieval arguments since we know

that only the newly inserted record will have the value of the @@Now variable for each of the above mentioned columns.

```

/* Retrieve auto-generated OrderID*/
SELECT @OrderID = OrderID
FROM Orders
WHERE OrderDate      = @@Now AND
      RequiredDate   = @@Now AND
      ShippedDate    = @@Now

/* check for error*/
IF (@@ERROR <> 0)
    GOTO Error

```

Finally, all that remains is to set OrderDate, RequiredDate, and ShippedDate to the correct value.

```

/*Update columns used in marker with proper value*/
UPDATE Orders SET
    OrderDate      = @OrderDate,
    RequiredDate   = @RequiredDate,
    ShippedDate    = @ShippedDate
WHERE OrderID = @OrderID

/* check for error*/
IF (@@ERROR <> 0)
    GOTO Error

GOTO Ok

Error:
/* Error Condition, so rollback*/
ROLLBACK TRANSACTION
RETURN -1
GOTO Finally

Ok:
/*Everything is ok, so commit*/
COMMIT TRANSACTION
RETURN 0

```

Finally:

END

GO

The sp_DeleteOrders Stored Procedure

The stored procedure for delete is much simpler than the other two:

```
-- =====
-- Script for sp_DeleteOrders
-- =====

/*****
/* Check if a version of the stored      */
/* procedure already exist and if so drop it */
*****/
IF EXISTS (SELECT name
           FROM   sysobjects
           WHERE  name = N'sp_DeleteOrders'
           AND   type = 'P')
    DROP PROCEDURE sp_DeleteOrders
GO

CREATE PROCEDURE sp_DeleteOrders
    @OrderID integer
AS
BEGIN
/*****
/* First delete order entries for the order */
/* to avoid constraint violation error.      */
*****/

BEGIN TRANSACTION

DELETE FROM [Order Details]
WHERE [Order Details].OrderID = @OrderID

/* check for error*/
IF (@@ERROR <> 0)
    GOTO Error
```

```

/* We can now delete the order from the Orders table */
DELETE FROM Orders
WHERE Orders.OrderID = @OrderID

/* check for error*/
IF (@@ERROR <> 0)
    GOTO Error

GOTO Ok

Error:
/* Error Condition, so rollback*/
ROLLBACK TRANSACTION
RETURN -1
GOTO Finally

Ok:
/*Everything is ok, so commit*/
COMMIT TRANSACTION
RETURN 0

Finally:

END

GO

```

Note that the script is designed in such a way to optimize the delete process. The stored procedure will delete the whole order, including the related rows in the Order Details table, to prevent foreign key constraint violation in the database.

The UpdateOrders Method

First, we need to declare the UpdateOrders method:

```

Protected Function UpdateOrders _
    (ByRef OrdersTB As DataTable) _
    As Integer

    'Declare SqlConnection object for connection to database
    'Use the Global Private string initialized in
    'MyBase.New()

```

```
Dim objConn As New SqlConnection(SQLConnectionString)

'Declare the DataAdapter that will be
'used to populate the DataSet.
Dim OrderDataAdapter As New SqlDataAdapter()

OrderDataAdapter.UpdateCommand = New SqlCommand()
```

Then we have to define the command text for UpdateCommand. We are going to use the stored procedures that we created. As you can see below, all you need to specify in the command text is the stored procedure name. We must also tell UpdateCommand that the commandtype is a stored procedure.

```
OrderDataAdapter.UpdateCommand.CommandText =
    "sp_UpdateOrders"
OrderDataAdapter.UpdateCommand.Connection = objConn
OrderDataAdapter.UpdateCommand.CommandType = _
    CommandType.StoredProcedure
```

Next, we need to define all the parameters that the stored procedure uses. The return value from the stored procedure is also captured in a parameter. This is a simple matter of setting the parameter direction to `ParameterDirection.ReturnValue`.

```
'Declare variable to be used as a
'reference to Parameters
Dim workParm As SqlParameter

'return value
workParm = OrderDataAdapter.UpdateCommand.
    Parameters.Add("@return", SqlDbType.Int)
workParm.Direction = ParameterDirection.ReturnValue
```

Remember that the OrderID is read-only, so we choose the original value and ignore the current value in case it has been changed.

```
'parameter 1
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@OrderID", SqlDbType.Int)
workParm.SourceColumn = "OrderID"
workParm.SourceVersion = DataRowVersion.Original
```

For the other parameters, since they are updateable, we choose the current value to apply any change to the database.

```
'parameter 2
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@CustomerID", SqlDbType.NChar, 5)
workParm.SourceColumn = "CustomerID"
workParm.SourceVersion = DataRowVersion.Current

'parameter 3
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@EmployeeID", SqlDbType.Int)
workParm.SourceColumn = "EmployeeID"
workParm.SourceVersion = DataRowVersion.Current

'parameter 4
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@OrderDate", SqlDbType.DateTime)
workParm.SourceColumn = "OrderDate"
workParm.SourceVersion = DataRowVersion.Current

'parameter 5
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@RequiredDate", SqlDbType.DateTime)
workParm.SourceColumn = "RequiredDate"
workParm.SourceVersion = DataRowVersion.Current

'parameter 6
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@ShippedDate", SqlDbType.DateTime)
workParm.SourceColumn = "ShippedDate"
workParm.SourceVersion = DataRowVersion.Current

'parameter 7
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@ShipVia", SqlDbType.Int)
workParm.SourceColumn = "ShipVia"
workParm.SourceVersion = DataRowVersion.Current
```

```
'parameter 8
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@Freight", SqlDbType.Money)
workParm.SourceColumn = "Freight"
workParm.SourceVersion = DataRowVersion.Current

'parameter 9
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@ShipName", SqlDbType.NVarChar, 40)
workParm.SourceColumn = "ShipName"
workParm.SourceVersion = DataRowVersion.Current

'parameter 10
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@ShipAddress", SqlDbType.NVarChar, 60)
workParm.SourceColumn = "ShipAddress"
workParm.SourceVersion = DataRowVersion.Current

'parameter 11
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@ShipCity", SqlDbType.NVarChar, 15)
workParm.SourceColumn = "ShipCity"
workParm.SourceVersion = DataRowVersion.Current

'parameter 12
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@ShipRegion", SqlDbType.NVarChar, 15)
workParm.SourceColumn = "ShipRegion"
workParm.SourceVersion = DataRowVersion.Current

'parameter 13
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@ShipPostalCode",
        SqlDbType.NVarChar, 10)
workParm.SourceColumn = "ShipPostalCode"
workParm.SourceVersion = DataRowVersion.Current

'parameter 14
workParm = OrderDataAdapter.UpdateCommand. _
    Parameters.Add("@ShipCountry", SqlDbType.NVarChar, 15)
workParm.SourceColumn = "ShipCountry"
workParm.SourceVersion = DataRowVersion.Current
```

We are now ready to do the update, passing only changed rows to the update method.

```

Try
    'Open the Connection to database
    objConn.Open()
    'Next process updates.
    UpdateOrders = OrderDataAdapter.Update _
        (OrdersTB.Select( _
Nothing, Nothing, DataRowState.ModifiedCurrent))

Catch ConcurrencyErr As DBConcurrencyException
    Throw ConcurrencyErr

Catch Err As Exception
    Throw New ApplicationException( _
        "Exception encountered when executing command.",
        Err)

Finally

    'Close the Connection to database
    objConn.Close()

End Try

End Function

```

The DeleteOrders Method

The DeleteOrders method follows a pattern similar to UpdateOrders.

```

Protected Function DeleteOrders _
    (ByRef OrdersTB As DataTable) _
    As Integer

    'Declare SqlConnection object for connection to database
    'Use the Global Private string initialized in
    'MyBase.New()
    Dim objConn As New SqlConnection(SQLConnectionString)

    'Declare the DataAdapter that will be
    'used to populate the DataSet.
    Dim OrderDataAdapter As New SqlDataAdapter()

    OrderDataAdapter.DeleteCommand = New SqlCommand()

```

```

Dim workParm As SqlParameter

OrderDataAdapter.DeleteCommand.CommandText =
    "sp_DeleteOrders"
OrderDataAdapter.DeleteCommand.Connection = objConn
OrderDataAdapter.DeleteCommand.CommandType = _
    CommandType.StoredProcedure

'return value
workParm = OrderDataAdapter.DeleteCommand. _
    Parameters.Add("@return", SqlDbType.Int)
workParm.Direction = ParameterDirection.ReturnValue

'parameter 1
workParm = OrderDataAdapter.DeleteCommand. _
    Parameters.Add("@OrderID", SqlDbType.Int)
workParm.SourceColumn = "OrderID"
workParm.SourceVersion = DataRowVersion.Original

Try
    'Open the Connection to database
    objConn.Open()
    ' Next process updates.
    DeleteOrders = OrderDataAdapter.Update( _
        OrdersTB.Select( _
            Nothing, Nothing, DataViewRowState.Deleted))

Catch ConcurrencyErr As DBConcurrencyException
    Throw ConcurrencyErr

Catch Err As Exception
    Throw New ApplicationException( _
        "Exception encountered when executing command.",
        Err)

Finally

    'Close the Connection to database
    objConn.Close()

End Try
End Function

```

The big difference from UpdateOrders is the smaller number of parameters required.

The InsertOrders Method

The InsertOrders Method also follows the same pattern. However, the parameter direction for the OrderID is now set to ParameterDirection.Output because we will need this value in the DataSet to update related records in the Order Details table.

```
Protected Function InsertOrders _
    (ByRef OrdersTB As DataTable) _
    As Integer

    'Declare SqlConnection object for connection to database
    'Use the Global Private string initialized in
    'MyBase.New()
    Dim objConn As New SqlConnection(SQLConnectionString)

    'Declare the DataAdapter that will be
    'used to populate the DataSet.
    Dim OrderDataAdapter As New SqlDataAdapter()

    OrderDataAdapter.InsertCommand = New SqlCommand()

    OrderDataAdapter.InsertCommand.CommandText =
        "sp_InsertOrders"
    OrderDataAdapter.InsertCommand.Connection = objConn
    OrderDataAdapter.InsertCommand.CommandType = _
        CommandType.StoredProcedure
```

We also need to instruct UpdateCommand to update data in the DataSet using the return output parameter. This is useful because we will only know the true value of OrderID after we have done the insert. Furthermore, using foreignkey constraint, we can cascade this update to include related child rows in the Order Details DataTable.

```
'Instruct InsertComand to use outparameter to update
'the dataset.
OrderDataAdapter.InsertCommand.UpdatedRowSource = _
    UpdateRowSource.OutputParameters
```

All that remains now is to continue with the same pattern as with the other order update methods.

```
Dim workParm As SqlParameter
'return value
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@return", SqlDbType.Int)
workParm.Direction = ParameterDirection.ReturnValue

'parameter 1
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@OrderID", SqlDbType.Int)
workParm.SourceColumn = "OrderID"
workParm.Direction = ParameterDirection.Output

'parameter 2
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@CustomerID", SqlDbType.NChar, 5)
workParm.SourceColumn = "CustomerID"
workParm.SourceVersion = DataRowVersion.Current

'parameter 3
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@EmployeeID", SqlDbType.Int)
workParm.SourceColumn = "EmployeeID"
workParm.SourceVersion = DataRowVersion.Current

'parameter 4
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@OrderDate", SqlDbType.DateTime)
workParm.SourceColumn = "OrderDate"
workParm.SourceVersion = DataRowVersion.Current

'parameter 5
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@RequiredDate", SqlDbType.DateTime)
workParm.SourceColumn = "RequiredDate"
workParm.SourceVersion = DataRowVersion.Current

'parameter 6
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@ShippedDate", SqlDbType.DateTime)
workParm.SourceColumn = "ShippedDate"
workParm.SourceVersion = DataRowVersion.Current
```

```
'parameter 7
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@ShipVia", SqlDbType.Int)
workParm.SourceColumn = "ShipVia"
workParm.SourceVersion = DataRowVersion.Current

'parameter 8
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@Freight", SqlDbType.Money)
workParm.SourceColumn = "Freight"
workParm.SourceVersion = DataRowVersion.Current

'parameter 9
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@ShipName", SqlDbType.NVarChar, 40)
workParm.SourceColumn = "ShipName"
workParm.SourceVersion = DataRowVersion.Current

'parameter 10
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@ShipAddress", SqlDbType.NVarChar, 60)
workParm.SourceColumn = "ShipAddress"
workParm.SourceVersion = DataRowVersion.Current

'parameter 11
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@ShipCity", SqlDbType.NVarChar, 15)
workParm.SourceColumn = "ShipCity"
workParm.SourceVersion = DataRowVersion.Current

'parameter 12
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@ShipRegion", SqlDbType.NVarChar, 15)
workParm.SourceColumn = "ShipRegion"
workParm.SourceVersion = DataRowVersion.Current

'parameter 13
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@ShipPostalCode", _
        SqlDbType.NVarChar, 10)
workParm.SourceColumn = "ShipPostalCode"
workParm.SourceVersion = DataRowVersion.Current
```

```
'parameter 14
workParm = OrderDataAdapter.InsertCommand. _
    Parameters.Add("@ShipCountry", SqlDbType.NVarChar, 15)
workParm.SourceColumn = "ShipCountry"
workParm.SourceVersion = DataRowVersion.Current

Try
    'Open the Connection to database
    objConn.Open()

    'Next process updates.
    InsertOrders = OrderDataAdapter.Update( _
        OrdersTB.Select( _
            Nothing, Nothing, DataRowState.Added))

Catch ConcurrencyErr As DBConcurrencyException
    Throw ConcurrencyErr

Catch Err As Exception
    Throw New ApplicationException( _
        "Exception encountered when executing command.", Err)

Finally

    'Close the Connection to database
    objConn.Close()

End Try

End Function
```

The FullUpdateOrder Method

Now that we have completed all the methods required for updating the Orders and Order Details tables, we can move on to expose the update functions to the Web Service clients. For the web method, we will only need to expose one method that does all the updates. The advantage of doing one method is to reduce the number of calls that have to be made to the Web Service. The disadvantage is that we are passing a bigger set of data at once, and it is more difficult to recover from update errors.

The signature for our method is shown here:

Class Method Name	Web Service Method Name	Description
FullUpdateOrder	FullUpdateOrder	Parameter: ByRef OrderDS (DataSet) Return: String Return error message if any

The first step is not the declaration of variables, as is usually the case, but rather we first check if the given DataSet contains the two required Data Tables for the update. If it does, then we provide reference variables to them.

```
<WebMethod(> _
Public Function FullUpdateOrder _
    (ByRef Orders As DataSet) _
    As String

    'check that the Order table is available in the dataset
    If Not Orders.Tables.Contains("Orders") Then
        Return "Order table not in dataset"
    End If

    'check that the Order Details table is available in
    'the dataset
    If Not Orders.Tables.Contains("Order Details") Then
        Return "Order Details table not in dataset"
    End If

    'reference the two tables for easier access
    Dim OrdersTB As DataTable = Orders.Tables("Orders")
    Dim OrderDetailsTB As DataTable = Orders.Tables("Order
    Details")
```

We now need to make sure that there is a foreign key constraint between the two Data Tables with cascading update.

```
'Make sure a foreign key between parent table (Orders)
'and child table (Order Details) exists
'Create foreign key
Dim fk_orders As ForeignKeyConstraint = _
    New ForeignKeyConstraint("fk_orders", _
        OrdersTB.Columns("OrderID"), _
        OrderDetailsTB.Columns("OrderID"))
fk_orders.DeleteRule = Rule.Cascade
fk_orders.UpdateRule = Rule.Cascade

' Add new foreign key constraint
Try
    OrderDetailsTB.Constraints.Add(fk_orders)
Catch err As Exception
    'If cannot add constraint, then
    'it must already exist or some other
    'restrictions exist
    'We can still try to update though
End Try
```

The key is only vital if we are going to do inserts.

We are now ready to begin the updates. First, we take care of the deletes. To avoid concurrency errors in the DataSet, we delete the child records first. If we do not do so, the DeleteOrders method will still work. However, when we later run the DeleteOrderDetails method, it will fail because child records marked as deleted in the DataSet (due to cascade delete) will no longer exist in the database and a concurrency error would be thrown. In other words, the DataSet will be out of sync with the database.

```
'The first step is to do the deletion
'Remember that must delete child first
'to avoid concurrency issue in dataset
Try
    DeleteOrderDetails(OrderDetailsTB)

Catch Err As Exception
    Return "Error deleting rows in OrderDetails,
        Message:" Err.Message
End Try
```

```
Try
  DeleteOrders(OrdersTB)
  Catch Err As Exception
    Return "Error deleting rows Orders, Message:" &
      Err.Message
End Try
```

Next, we do the inserts and the updates:

```
'Do Inserts
'With insert, the parent has to be inserted first
'to avoid concurrency error
Try
  InsertOrders(OrdersTB)
Catch Err As Exception
  Return "Error inserting rows Orders, Message:" &
    Err.Message
End Try

Try
  InsertOrderDetails(OrderDetailsTB)
  Catch Err As Exception
  Return "Error inserting rows in OrderDetails,
  Message:" & Err.Message
End Try

'Finally do the updates
Try
  UpdateOrders(OrdersTB)
Catch Err As Exception
  Return "Error updating rows Orders, Message:" &
    Err.Message
End Try

Try
  UpdateOrderDetails(OrderDetailsTB)
Catch Err As Exception
  Return "Error updating rows in OrderDetails,
  Message:" & Err.Message
End Try
Return ""

End Function
```

The method will return error messages if any exceptions are thrown. Otherwise, an empty string is returned.

Testing the Update Methods

Since these methods take DataSet as a parameter, the Web Server cannot automatically generate test pages. To do testing, you have to either create test methods, which you then comment out when you move from production to live system, or you can create a test application that uses the Web Service.

In the source code on the companion CD, you will find some test methods that you can use. Also included are additional methods that return employees and products.

Summary

The methods we developed for updating via DataSet and DataAdapter in this chapter are simple but form the basis on which you can build. Remember that we have only concentrated on the data access and update. In a commercial system, you will also need to consider other issues, such as performance, advance concurrency management, error recovery, and security.

Part III

Special Topics

Chapter 8

Migrating ADO Applications

In This Chapter

When you have a new version of a programming language, the biggest issue is whether migration of an existing application will bring any added benefits. Generally, the main reason for migration is for improved performance or added features that the new tool brings to the development process.

In this chapter, you will learn about different issues that you need to consider before you choose to migrate any existing ADO application to ADO .NET.

Legacy of Time

If you are not new to programming, you probably have, over time, written a fair amount of code. Even if you are relatively new to programming, you might have inherited code that you need to maintain and change. The legacy of time is such that we inherit many components and application frameworks that we need to maintain. Some code becomes redundant as new features are incorporated directly into the programming language, whereas other code is so specific to a particular need that it goes through much iteration and upgrade to improve performance and add additional capability.

If your main development platform is Microsoft, it is also likely that many of the components are in the form of DLLs or COM components. If these applications need access to a database of some sort, it is more than likely that you have a fair bit of code using ADO.

Language Changes

Before embarking head-on into converting your code to .NET, there are a few changes in the language syntax and structure of which you must be aware. If you are migrating an ASP application, you must keep in mind that VBScript has been replaced with VB .NET. Considering that VB .NET is practically a new language when compared to Visual Basic and VBScript, the makers of .NET have done a very good job of ensuring backward compatibility with existing Visual Basic applications. The support, however, is not 100%. There are a few considerations that you must take into account:

- **Data type variant:** The variant data type has been replaced with the object data type. The object data type must be explicitly cast to another data type before it can be used.
- **Method calls:** All method calls, regardless of the number of parameters, must now use parentheses. This is true even when there is no parameter.

```
Me.Close()
```
- **Arguments:** The default for passing arguments is now by value, as opposed to by reference. To pass arguments by reference, you must now use the `ByRef` keyword.
- **Object assignment:** `SET` and `LET` keywords are no longer valid. You can now use the assignment operator.

```
AnObject = AnotherObject
```

- **Default property:** To set the default property, you must explicitly reference the property. Previously, this was optional. However, the indexed default properties are still supported. For example, the Fields property, a default collection property of the RecordSet, does not have to be explicitly referenced.

```
'[Visual Basic]
Dim StrObjectName as AnObject = AnotherObject

RS("CustomerID").Value = "VINET"

'[VB.NET]
AnObject.Name = AnotherObject.Name

Dim StrObjectName as AnObject = AnotherObject.Name

'This line is still supported
RS("CustomerID").Value = "VINET"

'and is equivalent to
RS.Fields("CustomerID ").Value = "VINET"
```

- **Integer and long:** The integer data type is now 32 bits and the long data type is 64 bits.
- **Lazy evaluation:** VB .NET now uses lazy evaluation for Boolean expressions. This means that as soon as the Boolean expression can be evaluated, the expression is not processed further. So, in an AND expression, as soon as a false value is found, the expression is no longer parsed any further and the whole expression is evaluated to False. In the case of an OR expression, as soon as a true value is found, the evaluation terminates and the whole expression is evaluated to True. This is done for speed, but you must remember that if you depend on side effects of certain functions that return Boolean values, you

must nest the expression instead of using AND or OR expressions.

- **Explicit casting:** If you need to convert from one data type to another, you must explicitly cast the data type. For example, if a string is expected, it casts another data type as a string.

```
Response.Write("Employees ID is: " &
    CStr(IntEmployeeID))
```

- **Dim statement:** Variables within the same Dim statement will be of the same type.

```
[Visual Basic]
'A and B is a variant and only C is integer
DIM A, B, C As integer
```

```
[VB.NET]
'A, B and C are integer
DIM A, B, C As integer
```

- **Class property syntax:** The syntax no longer includes Property Get and Property Set. The new property syntax is similar to that in C#.

```
[Visual Basic]
Public Property AProperty As String
    Get
        aProperty = APrivateVariable
    End Get
    Set
        APrivateVariable = value
    End Set
End Property
```

```
[VB.NET]
Property AProperty( ) As String
    Get
        Return APrivateVariable
    End Get
    Set(ByVal Value As String)
        APrivateVariable = value
    End Set
End Property
```

- **& operator:** When using the & operator to concatenate strings, spaces must always be included.

```
[Visual Basic]
'No space required between & and variables
AllString = String1&String2&String3
```

```
[VB.NET]
'Must have space between & and variables
AllString = String1 & String2 & String3
```

- **If statements:** All If statements must be on multiple lines and end with End If.

```
[VB Script]
IF X Then Y
```

```
[VB.NET]
If X Then
Y
End If
```

What about COM?

With ASP web applications, the only way to compile and encapsulate business logic was to use COM components. COM components are also used in Visual Basic to add additional features and functions. Let's look at how .NET helps with migration of applications using COM components.

Most COM components that work in ASP will work in ASP .NET. Late-bound calls are still supported using `Server.CreateObject`; however, for better performance, it is recommended that early-bound calls are used. Before you can use a COM component in .NET, you must first expose the component in .NET assemblies. With Visual Studio.NET, this is very easy: You simply add a COM reference to your project, and the rest is automatically taken care of for you. If you are only using the .NET SDK, you can use the Type Library Importer

(TlbImp.exe). Type Library Importer converts standard COM components to the equivalent .NET Framework interoperations (InterOp) assemblies by building managed wrappers around the components.

.NET Framework Bidirectional Migration Support

So far, we have seen how to allow .NET applications to use existing COM components. What if we need to migrate the COM component itself but still want to use it in existing ASP and Visual Basic applications? Can .NET help? The answer to this is, of course, yes. The .NET InterOp services offer bidirectional support. It means that .NET components through InterOp service can be exposed as COM components. The System.Runtime.InteropServices namespace provides three categories of interop-specific attributes that you can use. However, I will not go into more detail because this is beyond the scope of this book.



Note: It is far easier to expose COM to .NET than to expose .NET as COM.

ASP and ASP .NET

Since ASP and ASP .NET uses different run times on the same server, the two can co-exist even within the same web application. This is usually while migration is in progress. You can slowly convert the ASP pages to ASP .NET, and the users can immediately get the benefits of the change.

What about ADO?

As far as .NET is concerned, ADO is simply a COM component. That means that all access to ADO is done through .NET COM InterOp. ADO objects and ADO

.NET are mutually exclusive. As a result, ADO and ADO.NET can exist together in the same application.

The best way to illustrate this is through an example. The easiest example to look at is an ASP application.

[ASP]

```

<%@LANGUAGE=VBSCRIPT%>
<!
This ASP example uses ADO to read records from the
Northwind database and print two fields from all returned
records to an ASP page. Connection to the Northwind
database is through an ODBC system data source.
>
<html>
<body>
<%
dim ADOconn, ADORecSet, SelectOrderSTR

SelectOrderSTR = _
    "SELECT OrderID, CustomerID, " & _
    "EmployeeID, OrderDate, " & _
    "RequiredDate, ShippedDate, " & _
    "ShipVia, Freight, " & _
    "ShipName, ShipAddress, " & _
    "ShipCity, ShipRegion, " & _
    "ShipPostalCode, ShipCountry " & _
    "FROM Orders with (NOLOCK)"

    set ADOconn = Server.CreateObject("ADODB.Connection")
    ADOconn.Open "DSN=NorthWindODBC;UID=sa;PWD=sa;"

    set ADORecSet = ADOconn.execute(SelectOrderSTR)
    ' Query didn't return any records.
    if ADORecSet.BOF and ADORecSet.EOF then
        Response.Write("No Records.")
    else
        'Query didn't return any records..MoveFirst
        Do While Not ADORecSet.EOF
            Response.Write(ADORecSet ("CustomerID") & " : " & _
                & ADORecSet ("OrderID") & "<br>")
            ADORecSet.MoveNext

```

```

        Loop
        Response.Write("<p>End of data.")
    end if
    ADORecSet.close
    set ADORecSet = nothing
%>
</body>
</html>

```

The minimum required to convert the above to ASP .NET is shown below. The file can be created simply by changing the extension of the ASP file to aspx.

[ASP .NET]

```

<%@Page aspcompat=true Language = VB%>
<!
This ASP example uses ADO to read records from the
Northwind database and print two fields from all returned
records to an ASP page.
Connection to the Northwind database is through an ODBC
system data source.
>
<html>
<body>
<%
dim ADOconn, ADORecSet, SelectOrderSTR

SelectOrderSTR = _
    "SELECT OrderID, CustomerID, " & _
    "EmployeeID, OrderDate, " & _
    "RequiredDate, ShippedDate, " & _
    "ShipVia, Freight, " & _
    "ShipName, ShipAddress, " & _
    "ShipCity, ShipRegion, " & _
    "ShipPostalCode, ShipCountry " & _
    "FROM Orders with (NOLOCK)"

'Set is removed
ADOconn = Server.CreateObject("ADODB.Connection")
'Parentheses added
ADOconn.Open("DSN=NorthWindODBC;UID=sa;PWD=sa;")

'Set is removed

```

```

ADOREcSet = ADOconn.execute(SelectOrderSTR)
' Query didn't return any records.
if ADORecSet.BOF and ADORecSet.EOF then
    Response.Write("No Records.")
else
    'Query didn't return any records..MoveFirst
    Do While Not ADORecSet.EOF
        ' Specify Value property.
        Response.Write(ADOREcSet ("CustomerID").value
            & " : " & ADORecSet ("OrderID").value & "<br>")
        ADORecSet.MoveNext
    Loop
    Response.Write("<p>End of data.")
end if
ADOREcSet.close
'Set is removed
ADOREcSet = nothing
%>
</body>
</html>

```

As is clearly shown, the main changes are to make the application conform to the new VB .NET syntax. Also, notice that the page directives attribute, `aspcompat`, is set to `True`. This is required so that the compiler knows that the page will include ASP type syntax and directives such as `<% %>` blocks for server scripts. This is the absolute minimum required. Also notice that in the example, the `<%@Page` does not specify any code-behind file.

In the example, we are using late binding, but for performance reasons it is recommended that you use early binding. You can only use early binding when you use compiled code, and for ASP .NET, this means using code-behind files. Note that this is the default for ASP .NET.

To use early binding after you have created the `aspx` file, you need to add the ADO reference. From within your project, go to the Add Reference dialog box and choose the COM tab. Look for the latest version of Microsoft

ActiveX Data Objects library. In our example, I will use version 2.7, so I choose Microsoft ActiveX Data Objects 2.7 Library. See Figure 8-1.

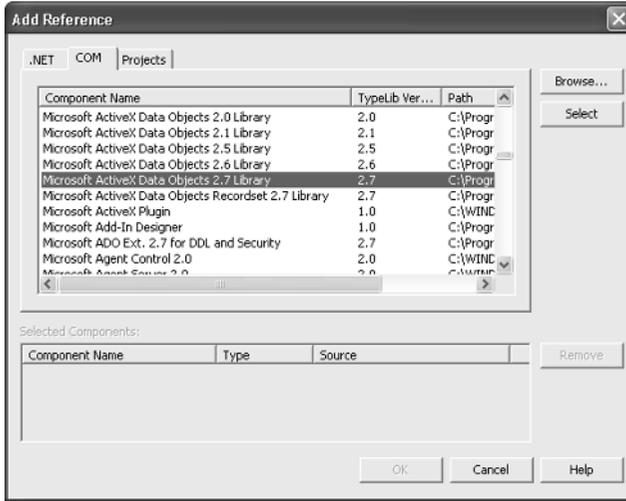


Figure 8-1: The Add Reference dialog

Once you click on Select and OK, Visual Studio .NET will add a reference to your project with a namespace called ADODB. See Figure 8-2.

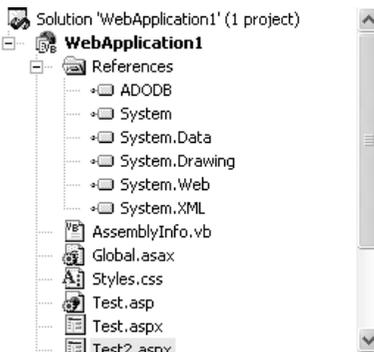


Figure 8-2: ADODB namespace added to the project

The ADODB namespace will now contain all the objects and associated methods that ADO uses. The next step is

to create a method in the class that we can use in the HTML section of the aspx file. We can still use the same example that we have used so far. We encapsulate this in a method called GetData.

```
Public Sub GetData()  
    'Create new ADO connection  
    Dim ADOconn As New ADODB.Connection()  
  
    'Declare Record Set  
    Dim ADORecSet As ADODB.Recordset  
  
    'Declare string variable  
    Dim SelectOrderSTR As String  
  
    SelectOrderSTR = _  
        "SELECT OrderID, CustomerID, " & _  
        "EmployeeID, OrderDate, " & _  
        "RequiredDate, ShippedDate, " & _  
        "ShipVia, Freight, " & _  
        "ShipName, ShipAddress, " & _  
        "ShipCity, ShipRegion, " & _  
        "ShipPostalCode, ShipCountry " & _  
        "FROM Orders with (NOLOCK)"  
  
    ADOconn.Open("DSN=NorthWindODBC;UID=sa;PWD=sa;")  
  
    'Set is removed  
    ADORecSet = ADOconn.Execute(SelectOrderSTR)  
    ' Query didn't return any records.  
    If ADORecSet.BOF And ADORecSet.EOF Then  
        Response.Write("No Records.")  
    Else  
        'Query didn't return any records..MoveFirst  
        Do While Not ADORecSet.EOF  
            'Specify Value property.  
  
            Response.Write(ADORecSet("CustomerID")  
                .Value & " : " & ADORecSet("OrderID")  
                .Value & "<br>")  
            ADORecSet.MoveNext()  
        Loop  
        Response.Write("<p>End of data.")  
    End If
```

```
ADORecSet.Close()  
'Set is removed  
ADORecSet = Nothing  
End Sub
```

The main difference with our previous example is that we are using late binding and we no longer need to use `Server.CreateObject`. The IntelliSense features of Visual Studio can now help with the programming and provide compile-time checking and strongly typed data types. However, we are not done yet. The final step is to use the method in the web page. To do this, you need to add a line in the HTML view of the aspx file.

```
<%@ Page Language="vb" AutoEventWireup="false"  
Codebehind="Test2.aspx.vb"  
Inherits="WebApplication1.Test2"%>  
<HTML>  
<! This ASP example uses ADO to read records from the  
Northwind database and print two fields from all  
returned records to an ASPX page. Connects to Northwind  
database through an ODBC system data source. >  
<body>  
<%  
    GetData()  
>%>  
</body>  
</HTML>
```

As you have probably gathered, .NET offers very good support for migration. In the case of ASP .NET, ASP and ASP .NET pages can even coexist in the same application. This makes it easier to migrate an application in different stages. As for Visual Basic applications, as long as you remember the few syntax changes, the migration process is simple.

To Migrate or Not to Migrate?

The biggest question now is whether or not to migrate. The answer to this question will depend on various factors, but keep in mind the old saying “If it ain’t broke, don’t fix it.”

Does the application lack certain desirable features that .NET can bring? If so, the application is a candidate for migration; however, this will still depend on other factors.

Are the application layers well-partitioned? In other words, are the presentation layer, business layer, and data layer well-partitioned? If the application is not well-designed, migration will be more difficult. It might be better to design a new system from scratch than to migrate what in effect is a badly designed system. This has to be weighed against cost and time.

Is data communication with other platforms or over the Internet with other applications required? XML and disconnected data connection is a breeze with ADO .NET.

Is the current client load overwhelming or will it soon overwhelm the current system? This is particularly true with e-commerce and business-to-business applications. .NET has various features that allow developers to build distributed systems that lend themselves very well to scaling and load balancing. Furthermore, ASP .NET has many performance enhancements over ASP.

Do you have the required support staff? .NET is relatively new, and even if you are an old hand with Visual Basic, there are many new things in VB .NET that you will need to learn and master. It might be worthwhile to delay the migration until your staff has acquired the

required level of expertise and experience. This book will go some way in helping you along that track.

Once you have considered the various factors, the next stage is to plan the migration.

Migration Steps

The prime candidates for migration are usually ASP web applications. This is because ASP .NET is a great leap forward in both design and run-time implementation of web applications. It provides various tools and enhancements that many web application developers wanted in the past. So from this point of view, we will consider mainly ASP applications, but the steps mentioned will also apply to Visual Basic.

Step 1: Migrate the Clients

Migrating the clients or presentation layer is probably the easiest step. This will allow you to build up experience and get most of the mundane work out of the way. The client can include both ASP pages and Windows Forms. Access to the business layer, which previously was done through COM, can still be maintained by using the InterOp services.

Step 2: Create .NET Wrappers to COM Components

Once the new clients are implemented and tested, the next step is to create .NET wrapper classes to the COM components. These are simply .NET classes that act as an interface to the COM components. The logic is still implemented in COM, but the clients now need to be changed so that they only access the .NET classes. At this stage the clients should be fully .NET.

Step 3: Migrate the Business Objects

Now you can go ahead and migrate all the COM components to .NET classes. At this stage, changes to the application will not affect the clients. Once this step is over, the application will be fully .NET.

Summary

In this chapter, we discussed the various issues that you need to consider when migrating applications to .NET. We concentrated mainly on COM and ADO because they are more relevant to this book. There are many more issues to consider when you migrate an application. For more details, you can look in the help files. A good article is included that can be accessed from <ms-help://MS.VSCC/MS.MSDNVS/dnaspp/html/aspnetmigrissues.htm> in the Visual Studio .NET help file.

Manipulating Multidimensional Data

In This Chapter

This chapter is a case study that covers the design and implementation of an OLAP solution in Visual Studio .NET utilizing the industry's premier OLAP Microsoft SQL Server Analysis Services as the data store.

As an in-house systems developer, you realize that your company is bracing for an era of data revolution. This is at a time when information is the key to the success of your organization and its business processes. The marketers would like to analyze the trend of customer behavior, while the salespeople want to keep track of sales activities over a long period of time. Such information is an imperative part of plotting your company's future marketing and production strategy. As a systems developer, you are struggling to design an effective and scalable software solution that your company's analysts can use to assess its performance in the marketplace.

You quickly realize that implementing custom reports based on the company's day-to-day data processing system would simply create a bottleneck as more stress is imposed on the databases. Thus, you revert to the design of an OLAP data warehouse.

A Quick Primer on Analysis Services

OLAP stands for online analytical processing. In a typical OLAP data warehouse, there is an OLAP server that stores data and a client tool that users can utilize to perform analysis of the stored data. The multidimensional storage format is very different from the relational data storage that you are used to. It ensures that the proper relations are built between database entities, and it formats the data in a way that is easy and efficient to analyze.

SQL Server 2000 Analysis Services is Microsoft's powerful OLAP data warehouse solution architecture that you can use to provide real-time corporate performance analysis.

Before we go over the development of the OLAP solution, let's take a quick walk through its installation and identify its key features. This chapter does not turn you into an Analysis Services expert. To obtain a much deeper insight into the technology, you may want to read *Professional SQL Server 2000 Data Warehousing with Analysis Services*.

Analysis Services Installation

This section focuses on the installation of Microsoft SQL Server 2000 Analysis Services.

System Requirements

System Hardware Requirements

Table 9-1 is a list of all the hardware requirements for the installation of SQL Server 2000 Analysis Services.

Table 9-1: Hardware requirements for SQL Server 2000 Analysis Services

Hardware Component	Minimum Requirement
Processor	Type: Intel Pentium I or higher Speed: 135 MHz
Memory	64MB
Disk Storage	Server: A minimum server installation requires 50MB. The full server installation requires up to 130MB. Client: The full installation of client components requires 12MB.

System Software Requirements

Table 9-2 is a list of all the software requirements for the installation of SQL Server 2000 Analysis Services.

Table 9-2: Software requirements for SQL Server 2000 Analysis Services

Software Component	Minimum Requirement
System Software	Server: For any of the following Windows operating systems, the OS must not be a domain controller: Windows NT Server 4.0, SP5 + Windows 2000 Server Client: Any of the following Windows operating systems: Windows 95 with DCOM 5 Windows 98 Windows NT Workstation 4 Windows 2000 Professional Windows XP Professional
Other Software	Internet Explorer 5+ is a required component for the administration of the server. Install this software on the client or server from which you would want to manage the OLAP server.

Installation Components

Analysis Services comes with several modularized components. This provides you with the flexibility of installing only the components that you want to work with.

Table 9-3 is a list of the components that can be installed.

Table 9-3: The installation components

Component to Install	Component Type	Component Storage Requirement	Description of Component
SQL Server 2000 Analysis Server	Server component	34140KB	A group of executables and other files that make up the Analysis Services Server. It is only installed on the server machine and is required for an OLAP solution.
SQL Server 2000 Analysis Manager	Client-server component	35304KB	A group of files that provides the developer with an intuitive GUI for manipulating and administering objects on the Analysis Server. This component can be installed on the server, the client, or both. Microsoft also provides a web-based administration tool for remote administration.
Decision Support Objects (DSO)	Client-server component	7128KB	DSOs are a group of COM+-based components that have very sophisticated object models for the purpose of custom server administration and management of metadata processes.

Component to Install	Component Type	Component Storage Requirement	Description of Component
PivotTable Service	Client component	9776KB	The PivotTable Service installs the OLEDB Provider for analysis services (MSOLAP) and ADO MD (ActiveX Data Objects Multidimensional) that provide all the means of communication between the database server and the client applications. The OLEDB Provider for Analysis Services is the component used in the .NET Framework to access OLAP databases.
Sample Applications	Client component	2772KB	Sample applications that demonstrate the use of Analysis Services. For our case study, we will use the sample FoodMart 2000 database.
Books Online	Client-server component	29844KB	Books Online is a very comprehensive help file featuring a full reference to SQL Server 2000 Analysis Services.

As you may have noticed, Analysis Services comes with a lot of components. Simply knowing what the components are and where to install them, however, does not help in performing the right installation. Next, we will walk through the installation process of Analysis Services.

Setup

The setup of SQL Server 2000 Analysis Services is quite straightforward. However, you do need to know exactly what, where, how, and why to install. This section provides a complete walk-through of Analysis Services installation.

The following steps are provided to guide you through the installation process. At every step, you are introduced to the installation issues, which are explained in great detail.

Starting Up

1. Insert the SQL Server 2000 Setup CD into your CD-ROM drive.
2. From the **Start** menu, click **Run**. This loads the Windows Run dialog.
3. On the Windows Run dialog, click the **Browse** button.
4. In the Browse dialog that is loaded, locate and open the **Autorun.exe** file on the Setup CD. It is usually located in the root directory. If autorun is enabled on your computer, Autorun.exe will run automatically and you can skip all these steps.
5. Click **OK** on the Run dialog. The SQL Server 2000 Welcome dialog should pop up.

Running Setup

1. To start the installation, click the **SQL Server Components** option on the Welcome dialog. The Install Components dialog opens, as illustrated in Figure 9-1.



Figure 9-1: The Install Components dialog for the SQL Server 2000 Developer Edition

2. The Install Components dialog gives you three options:
 - **Install Database Server:** This option installs an SQL Server.
 - **Install Analysis Services:** This option installs the Analysis Services.
 - **Install English Query:** This option installs the Microsoft English Query.
3. Click the **Install Analysis Services** option to start the InstallShield Setup Wizard.
4. You are presented with a Welcome dialog that introduces you to Microsoft Analysis Services. Click the **Next** button.
5. After reading the license agreement, click the **Yes** button on the Software License Agreement dialog.
6. The Select Components dialog is the most important dialog in the installation of Analysis Services. You are given the options for installing the components that

were discussed in Table 9-3. The Select Components dialog is illustrated in Figure 9-2.

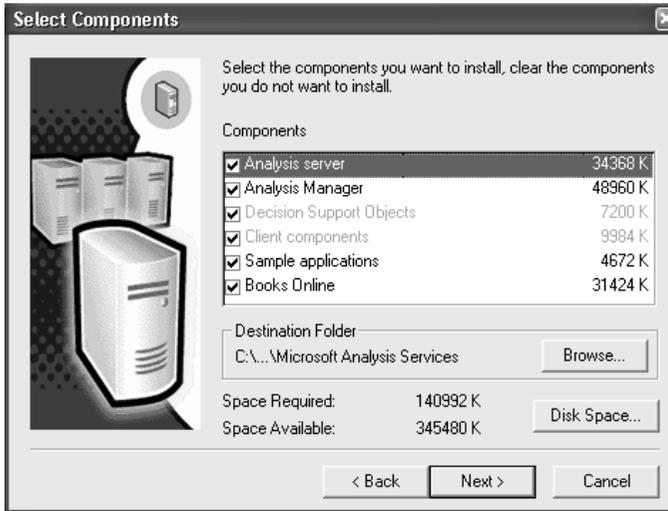


Figure 9-2: The Select Components dialog

7. In the Select Components dialog, specify which Analysis Services components you want to install.

To select a component:

- a. Click on that particular component and make sure that the check box to its left is selected.
 - b. The operation is vice versa for deselecting a selected component.
8. Although it is advisable to leave the default destination folders as they are, you may decide to install different components to different locations.

To specify the destination location of a component:

- a. Click on that particular component.
- b. Click the **Browse** button.
- c. In the Choose Folder dialog, select the destination folder into which you wish to install the

component. You can even specify a location that does not exist. In this case, the installation will create the location for you. After selecting a location, click the **OK** button on the Choose Folder dialog.

9. When you are ready to continue with the installation process, click the **Next** button.
10. For easier access to the Analysis Services components, the installation process provides you with an option to specify a program folder for the components. Do that in the Select Program Folder dialog illustrated in Figure 9-3.

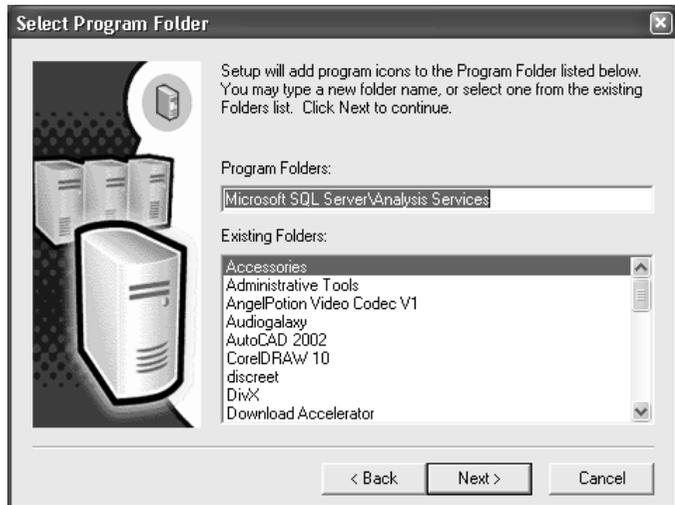


Figure 9-3: The Select Program Folder dialog

11. Click the **Next** button to start the setup process. The setup for Analysis Services may take a while, depending on the configuration of the machine on which it is being installed. The procedure for the setup usually starts by updating your machine with the latest MDAC components that are available on the Setup CD-ROM. Following that, Analysis

Services files are copied to your hard disk, and the necessary configuration settings are made to the Windows registry.

12. After the files are copied and the settings made, the setup procedure displays the Setup Complete dialog. Due to some changes that occurred with your configuration settings during setup, you are asked to restart your computer. You may choose the **Restart Now** option or you can restart later. However, it is very important to reboot the computer before using the Analysis Services. After making your selection, click the **Finish** button.

Understanding the Data Source

This section identifies the major low-level parts of Analysis Services and then focuses on its architecture for the purpose of our example. Analysis Services has two parts:

- The OLAP Server, which stores the data
- The PivotTable Service, which is a set of tools that allows data manipulation from the client

For our example, we are going to take a look at the sample FoodMart 2000 database that comes with Analysis Services. The general relationship of the client-server data manipulation architecture of Analysis Services is illustrated on the following page.

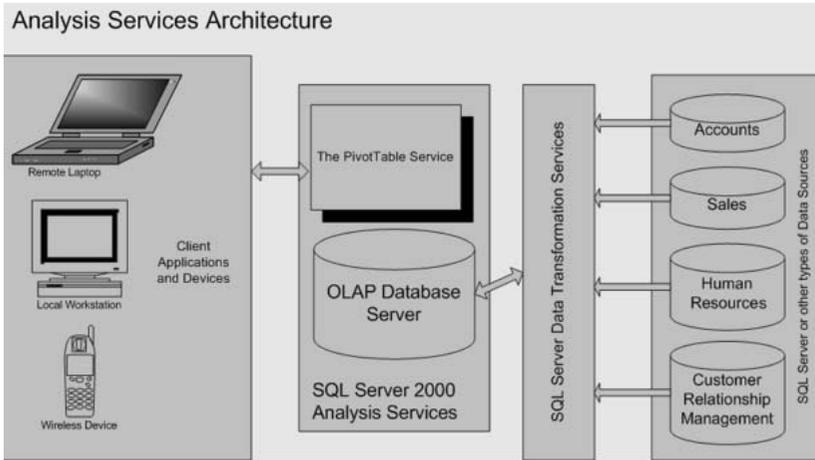


Figure 9-4: Analysis Services architecture

The Relational Database

The relational database for the FoodMart 2000 sample is implemented as a Microsoft Access application. It is located under the Samples folder, which is installed inside the Analysis Services installation directory. This path depends on where you originally specified the installation directory of Analysis Services. There are two files inside this directory that we are going to look at:

- **FoodMart 2000.mdb:** This is the actual relational database from which the OLAP database is created.
- **FoodMart 2000.cab:** This is an archive of the OLAP database provided by the installation process. We will restore this database after our discussion of the relational structure of the FoodMart 2000.mdb.

The FoodMart 2000 database keeps track of sales activities for a fictional company called FoodMart. Critical information about customers, products, employees, warehouses, and purchases are kept inside the database. It is not our goal to examine the exact design of the database, so I will not go into a detailed discussion of the

database structure. Figure 9-5 provides a quick summary of the relational structure of the FoodMart database.

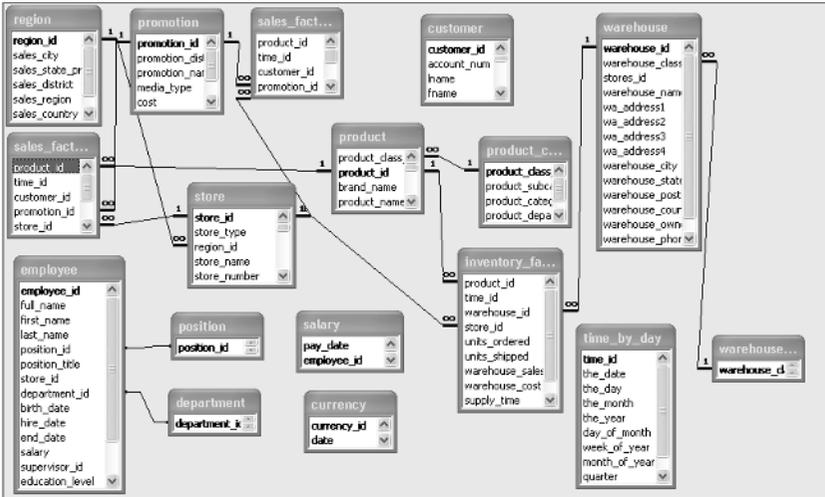


Figure 9-5: The relational structure of the FoodMart database

The OLAP Database

The OLAP Database Server stores data in a multidimensional format. The data is grouped into cubes that the database designer specifies.

Populating the OLAP Database

In general, an OLAP database is a warehouse database. It follows that the data must be available from a different source in order for it to be populated. In Figure 9-4, the OLAP database is using data from four separate databases: Accounts, Sales, Human Resources, and CRM. Data from these four databases is combined and stored in the OLAP database for future analysis. The data source from which to obtain data for an OLAP database does not necessarily have to be a relational data source. It can be any form of data that you want to analyze. This includes text files, Windows Active Directory, and Exchange Server, just to name a few.

How is the Data Stored?

The data in an OLAP database is stored inside cubes. A *cube* is simply a conceptual way of expressing a multidimensional storage model. OLAP cubes have three storage architectures: MOLAP, ROLAP, and HOLAP. For a more in-depth overview of multidimensional storage architecture, see *Professional SQL Server 2000 Data Warehousing with Analysis Services*.

PivotTable Service

The PivotTable Service is the client tool used to access, retrieve, and manipulate multidimensional data from the OLAP Server.

 **Note:** You cannot update an OLAP data source because it is simply an Analysis Server and nothing more. It is not designed to provide OLTP support.

The PivotTable Service has a very loose architecture that you must understand before attempting to program for Analysis Services. The PivotTable Service architecture has two main components to start with: the OLEDB Provider for OLAP and ADO MD. We will take an in-depth look at both of these technologies as we proceed through this chapter.

OLEDB Provider for OLAP

OLEDB Provider for OLAP is one of the thousands of OLEDB providers available on the market today. Microsoft developed the OLEDB Provider for OLAP MSOLAP driver to be used specifically with any OLAP database, although it is optimized for Analysis Services. OLEDB Provider for OLAP is to Analysis Services what the SQLOLEDB driver is to SQL Server. To manipulate OLAP data, developers can use ADO MD, implement their own consumers for OLEDB for OLAP, or use the provider's API in their code. Expect to see more

extensive .NET-enabled tools out with the next version of SQL Server Analysis Services.

Multidimensional Expressions (MDX)

MDX is the data manipulation language (DML) used for OLAP data sources. Its syntax is very close to that of its predecessor, SQL. The only difference between the two is that the latter can only manipulate the relational data source, while MDX is used in a multidimensional scenario. Therefore, MDX is simply an optimized version of SQL. OLEDB providers, such as SQLOLEDB, cannot process MDX queries. However, MSOLAP can process SQL queries, and it is possible to query OLAP data sources with SQL.

This section does not cover the entire MDX specification as released by Microsoft. We will walk through some simple syntax in our examples and explain the semantics of the language. Unless you have a good knowledge of cubic data storage, it is impossible to fully understand the MDX syntax. For a full discussion of the topic, see *Professional SQL Server 2000 Data Warehousing with Analysis Services*.

ActiveX Data Object Multidimensional (ADO MD)

For those who are reading about it for the first time, ADO MD may seem to be something new. In fact, it is not, at least if you are accustomed to ADO. ADO MD is an elegant COM component library that is simply an extended version of ADO (ActiveX Data Object). While Microsoft implemented ADO as a consumer of OLEDB, ADO MD is a consumer to OLEDB Provider for OLAP. It is virtually impossible to retrieve multidimensional data using ADO. The objects in ADO MD have been optimized to do this job efficiently and seamlessly.

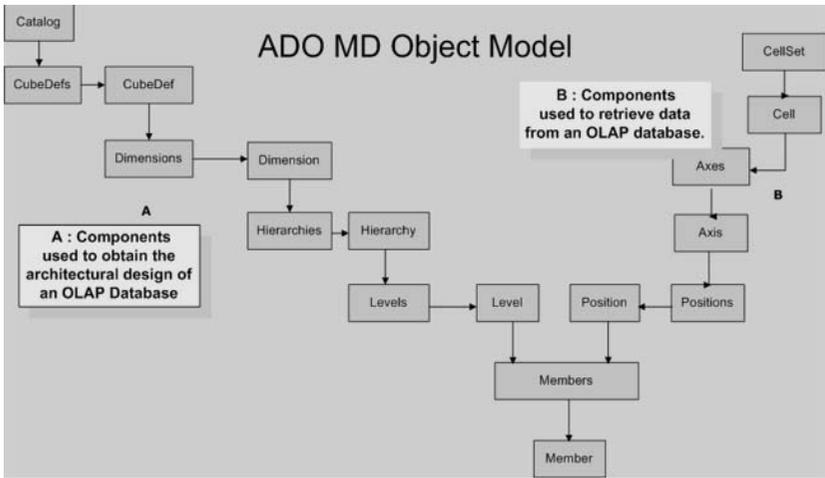


Figure 9-6: The ADO MD object model

Figure 9-6 provides you with a nice view of the ADO MD object model. In the next few sections, you will be reading and learning more about it. There is even an example of how to use it in your code to retrieve and manipulate OLAP data on the client application.

As I explain the concepts of ADO MD, I will also illustrate how to use it through the implementation of a small Windows Form application. The code is very easy and straightforward to follow. It is provided on the companion CD.

Visiting the Design of the Database

Through the ADO MD objects, it is possible for you to obtain every single detail of knowledge you require about the structure of an OLAP data source. It is possible to take a look at all the objects in the data source. For this small section, our focus is on Part A of Figure 9-6. We will discuss the route from the Catalog object to the Level object.

When operating on an OLAP database, the *Catalog object* represents the entire database. It has three properties: Name, ActiveConnection, and CubeDefs. The name of the Catalog object in our example would be FoodMart 2000. The properties and their purposes are illustrated in Table 9-4:

Table 9-4: Properties of the Catalog object

Property	Purpose
Name	Specifies the name of the OLAP database. This property is read-only.
ActiveConnection	This defines the connection string used to connect to the database. This property is read and write. Upon assigning a valid string value to it, the Catalog object automatically connects to the data source.
CubeDefs	This property is a collection of all the Cube objects referencing all the cubes that are available in the database referenced by the Catalog object.

A *Cube object* is a reference to one OLAP cube in the database. It is not possible to access a cube directly using the Cube object. You must have a valid Catalog object with a CubeDefs collection containing all the available cubes. The properties of the Cube object are discussed in Table 9-5.

Table 9-5: Properties of the Cube object

Property	Purpose
Name	Specifies the name of the OLAP cube. This property is read-only.
Dimensions	This is a collection of all the dimensions available for the particular cube.

The FoodMart 2000 database contains six cubes:

- Budget
- HR
- Sales
- Trained Cube
- Warehouse
- Warehouse and Sales

The *Dimension object* is one dimension of an OLAP cube. It can be instantiated using one item in the Dimensions collection of a Cube object. The properties of the Dimension object are discussed in Table 9-6.

Table 9-6: Properties of the Dimension object

Property	Purpose
Name	Specifies the name of the cube dimension. This property is read-only.
Hierarchies	This is a collection of all the hierarchies available for the particular dimension.

A *Hierarchy object* stands for a valid hierarchy of a dimension. It is initialized using an item from the Hierarchies collection property of a Dimension object. The properties of the Hierarchy object are discussed in Table 9-7.

Table 9-7: Properties of the Hierarchy object

Property	Purpose
UniqueName	Specifies the name of the cube dimension. This property is read-only.
Levels	This is a collection of all the levels available for the particular Hierarchy. It is important to note that while inside a database, levels have different positions and can be children of another level; this is not the case inside the Levels collection. Level objects maintain their child-parent state through one of their own properties.

In this example, you take a look at the Windows application that retrieves an OLAP database design.

This is the code for the form:

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Dim objCatalog As Object

    #Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()
        'This call is required by the Windows Form
        'Designer.
        InitializeComponent()
        'Add any initialization after the
        'InitializeComponent() call
    End Sub

    'Form overrides dispose to clean up the component
    'list.
    Protected Overloads Overrides _
    Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Windows Form Designer
    Private components As _
    System.ComponentModel.IContainer

    'NOTE: The following procedure is required by the
    'Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    Friend WithEvents Label1 As _
    System.Windows.Forms.Label
```

```

Friend WithEvents lblCatalogName As _
System.Windows.Forms.Label

Friend WithEvents lstCubes As _
System.Windows.Forms.ListBox

Friend WithEvents Label2 As _
System.Windows.Forms.Label

Friend WithEvents lstDimensions As _
System.Windows.Forms.ListBox

Friend WithEvents lstHierarchies As _
System.Windows.Forms.ListBox

<System.Diagnostics.DebuggerStepThrough()> _
Private Sub InitializeComponent()
    Me.Label1 = New System.Windows.Forms.Label()
    Me.lblCatalogName = New
        System.Windows.Forms.Label()
    Me.lstCubes = New System.Windows.Forms.ListBox()
    Me.Label2 = New System.Windows.Forms.Label()
    Me.lstDimensions = New
        System.Windows.Forms.ListBox()
    Me.lstHierarchies = New
        System.Windows.Forms.ListBox()
    Me.SuspendLayout()
    '
    'Label1
    '
    Me.Label1.Location = New System.Drawing.Point(8,
        16)
    Me.Label1.Name = "Label1"
    Me.Label1.Size = New System.Drawing.Size(80, 16)
    Me.Label1.TabIndex = 0
    Me.Label1.Text = "Catalog Name:"
    '
    'lblCatalogName
    '
    Me.lblCatalogName.Location = New _
System.Drawing.Point(88, 16)

```

```
Me.lblCatalogName.Name = "lblCatalogName"
Me.lblCatalogName.Size = New
    System.Drawing.Size(272, 23)
Me.lblCatalogName.TabIndex = 1
'
'lstCubes
'
Me.lstCubes.Location = New System.Drawing.Point(8,
    56)
Me.lstCubes.Name = "lstCubes"
Me.lstCubes.Size = New System.Drawing.Size(120,
    134)
Me.lstCubes.TabIndex = 2
'
'Label2
'
Me.Label2.Location = New System.Drawing.Point(8,
    40)
Me.Label2.Name = "Label2"
Me.Label2.Size = New System.Drawing.Size(100, 16)
Me.Label2.TabIndex = 3
Me.Label2.Text = "Cubes:"
'
'lstDimensions
'
Me.lstDimensions.Location = New _
System.Drawing.Point(136, 56)

Me.lstDimensions.Name = "lstDimensions"
Me.lstDimensions.Size = New
    System.Drawing.Size(120, 134)
Me.lstDimensions.TabIndex = 4
'
'lstHierarchies
'
Me.lstHierarchies.Location = New _
System.Drawing.Point(264, 56)
Me.lstHierarchies.Name = "lstHierarchies"
Me.lstHierarchies.Size = New
    System.Drawing.Size(120, 134)
Me.lstHierarchies.TabIndex = 5
'
'Form1
'
```

```

Me.AutoScaleBaseSize = New System.Drawing.Size(5,
13)
Me.ClientSize = New System.Drawing.Size(392, 195)

Me.Controls.AddRange(New
    System.Windows.Forms.Control() _
{Me.lstHierarchies, Me.lstDimensions, Me.Label2, _
Me.lstCubes, Me.tblCatalogName, Me.Label1})

Me.FormBorderStyle = _
System.Windows.Forms.FormBorderStyle.FixedTool-
Window

Me.Name = "Form1"
Me.Text = "Example 1"
Me.ResumeLayout(False)

End Sub

#End Region

Private Sub Form1_Load(ByVal sender As System.
Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
    ReadCatalog()
End Sub

Public Function Connect()
    Dim strConn As String

    strConn = "Data Source= LOCALHOST; Provider=
        MSOLAP;" _
        & "Database=FoodMart 2000;UserId=" _
        & "Password="
    'create the COM object
    objCatalog = CreateObject("ADOMD.Catalog")
    'Connect to the database
    objCatalog.ActiveConnection = strConn
End Function

Public Function ReadCatalog()
    Connect()
    tblCatalogName.Text = objCatalog.name

```

```

    GetCubes()
End Function

Public Function GetCubes()
    Dim Cube As Object
    For Each Cube In objCatalog.CubeDefs
        lstCubes.Items.Add(Cube.name)
    Next
End Function

Public Function GetDimensions()
    Dim Dimension As Object
    lstDimensions.Items.Clear()

    For Each Dimension In _
objCatalog.CubeDefs(lstCubes.SelectedItem).
        dimensions
        lstDimensions.Items.Add(Dimension.name)
    Next
End Function

Public Function GetHierarchies()
    Dim Hierarchy As Object
    lstHierarchies.Items.Clear()
    For Each Hierarchy In _
objCatalog.CubeDefs(lstCubes.SelectedItem).
        Dimensions( _
lstDimensions.SelectedItem).Hierarchies
        lstHierarchies.Items.Add(Hierarchy.UniqueName)
    Next
End Function

Private Sub lstCubes_SelectedIndexChanged
    (ByVal sender As _
System.Object, ByVal e As System.EventArgs) Handles _
lstCubes.SelectedIndexChanged
    GetDimensions()

End Sub

Private Sub lstDimensions_SelectedIndexChanged _
    (ByVal sender As System.Object, ByVal e As _

```

```

System.EventArgs) Handles _
    lstDimensions.SelectedIndexChanged

    GetHierarchies()
End Sub
End Class
    
```

Figure 9-7 shows the Visual Studio .NET Designer with Form1 in design view:

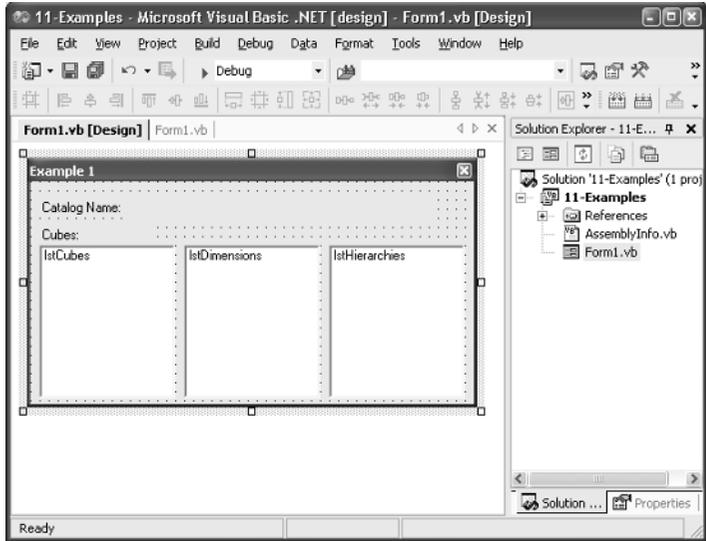


Figure 9-7: The Visual Studio .NET Designer

Figure 9-8 shows the output of the sample code and the Form1 object:

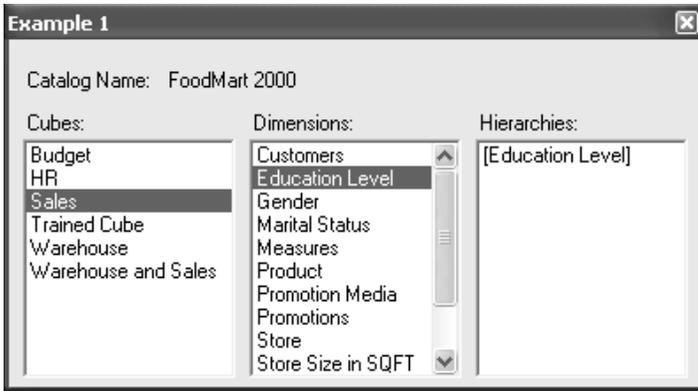


Figure 9-8: A view of the OLAP database containing the output of the Form1 object

You can select a particular cube from the Cubes list box to view a list of its dimensions in the Dimensions list box. Each dimension may have one or more hierarchies listed in the Hierarchies list box.

ADO MD Example

There is no ADO MD equivalent in ADO .NET. However, it is expected that Microsoft will release some ADO MD functionality that runs natively on the .NET Framework with the next version of Analysis Services. For now, the best way for us to continue programming Analysis Services in .NET is to utilize ActiveX controls provided for Analysis Services. Of course, you can use the OLEDB .NET Data Provider and connect to the database through the OLEDB Provider for OLAP. Even then, you will have to write a lot of unmanaged code that will have a direct impact on application performance since OLEDB Provider for OLAP runs on top of the PivotTable Service, which is implemented as COM.

There are several ActiveX controls deployed with Analysis Services. The one that is the most illustrative and helpful is the CubeBrowser ActiveX control.

Using the CubeBrowser ActiveX Control

In this example, we look at how we can use the CubeBrowser ActiveX control to view multidimensional data inside client applications. There are several advantages and disadvantages to using this control in your applications.

The main advantage of the CubeBrowser control is that it is straightforward and there is not much code needed to populate it with data. In fact, all you need to specify are the server that it should connect to and the cube that it should display. This saves you the trouble of writing a lot of unmanaged code that would have to go through the COM interoperability services.

There are two main disadvantages to using this control: First, keep in mind that the control is based on the COM computing architecture. Although you may not visually notice anything at design time, the .NET Common Language Runtime wraps the ActiveX control inside several COM interoperability service libraries when the application runs. Your application may end up having a lot of performance problems. Additionally, you will have a lot of trouble deploying your application, especially if it is a client-server application with multiple users. It would be required for you to deploy and register the CubeBrowser control on every client machine running the application. Without a sophisticated setup program, this can be quite a pain.

Although the disadvantages may seem to be enormous, I believe that using this control is a much better approach than using the OLEDB .NET Data Provider. As mentioned above, the control does not require you to write a

lot of code, which means that when the OLAP services are introduced inside ADO .NET, you would only need to delete the control, its references, and the other references to the COM interoperability service libraries. Then you can start writing managed code for the OLAP services. This task is much less arduous than having to convert unmanaged code.

Now that we have explored the pros and cons, let's get to work. In this example, you would be using the control on a Windows Form object. To view the code for this example, open the Example 2 folder under the 09-Examples Windows Form project on the companion CD and open the frmOLAPManip1 form object.

The first step is to add the control to your toolbox:

1. Open the **General** tab on the toolbox.
2. Right-click inside the tab and click the **Customize Toolbox** option on the pop-up menu. This loads the Customize Toolbox dialog.
3. The control is an ActiveX control and based on COM, so select the **COM Components** tab. The dialog populates a list of available ActiveX controls.
4. Scroll down the list and select the control called **OLAPCubeBrowser.CubeBrowser**, as illustrated in Figure 9-9.

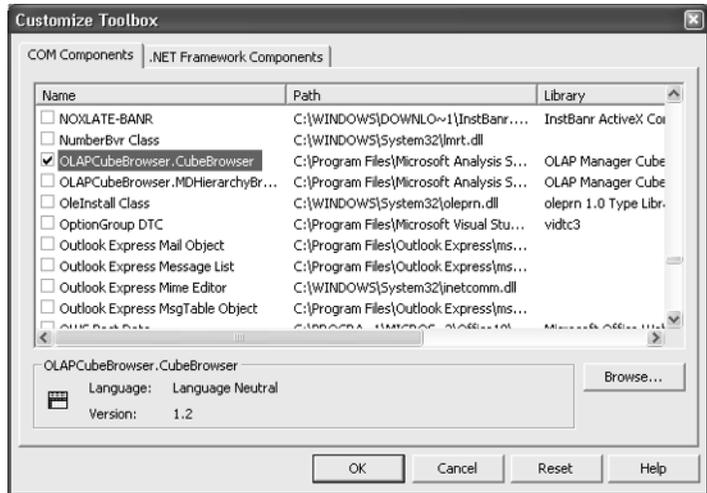


Figure 9-9: Selecting the OLAPCubeBrowser.CubeBrowser control

5. Click the **OK** button. The CubeBrowser control is added to the General tab of the toolbox.

The second step involves referencing ADO MD in your application:

1. From the **Project** menu, click **Add Reference**. This brings up the Add Reference window.
2. ADO MD is a COM library, so click on the **COM** tab to list all the COM libraries registered on your machine.
3. Click on the **Microsoft ActiveX Data Objects (Multi-dimensional) 2.7 Library** COM library. Click the **Select** button. This adds the library to the Selected Components list box, as illustrated in Figure 9-10.

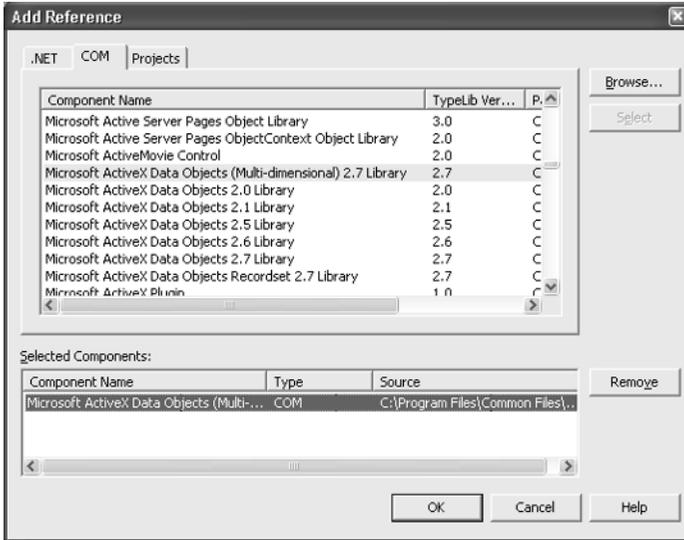


Figure 9-10: Adding a library to the Selected Components list box

4. Click the **OK** button. Visual Studio adds references to the following libraries:
 - **ADODB:** This is the legacy COM ADO library. It is referenced automatically because ADO MD requires ADO's Connection object to connect to data sources. Do not confuse ADO DB with the ADO .NET classes found inside the System.Data namespace.
 - **ADOMD:** This is the ADO MD class library that is required by the CubeBrowser control.

The third step involves adding the control to the form and writing some code.

1. Drag the CubeBrowser control from the General tab onto the frmOLAPManip1 form. Visual Studio adds references to the following libraries:
 - **AxOLAPCubeBrowser:** This is a wrapper for the CubeBrowser control. Visual Studio

automatically creates a wrapper for every ActiveX control used in an application. This allows your control to run in a managed .NET environment, and it also facilitates your job as a programmer since it exposes the control as a fully featured Windows Form Control. This is something that some developers pay no attention to. It is important for you to understand that this wrapper class library is going through COM interoperability services at run time.

- **MSComctlLib:** This is a COM class library that contains the implementation and definition of the Microsoft Windows Common Controls. Since some of these controls are used in the implementation of the CubeBrowser, this library is also automatically referenced.
- **OLAPCubeBrowser:** This is the class library for the CubeBrowser control.
- **stdole:** The COM architecture provides a standard implementation of general services, such as fonts, control locations, and positions of OLE components through interfaces. For example, a standard implementation of font objects is provided by the IFontDisp interface. The functionality is encapsulated by the StdFont object found within the stdole class library.

These interfaces are implemented inside the stdole class library. This library is automatically referenced in the background when each COM component is compiled. Therefore, it is referenced automatically by Visual Studio.

2. Open the Property window for the control and rename it **AxCubeBrowser**.

The fourth step involves adding a few tweaks to the user interface and writing the code that will manipulate the CubeBrowser.

The CubeBrowser and the ComboBox would need to connect to and retrieve data from the database. For this, we will need a global connection string, a connection object, and a procedure to connect to the database. Add the following declarations in the code module to the form:

```
Dim strConn As String
Dim objADODConn As Object
```

A procedure to connect to the database is implemented like this:

```
Sub Connect()
    'Connection string
    strConn = "Data Source= LOCALHOST; Provider=
        MSOLAP;" _
        & "Database=FoodMart 2000;UserId=" _
        & "Password="
    'Creates an ADO connection object.
    objADODConn = CreateObject("ADODB.Connection")

    With objADODConn
        'Sets the connection string
        .ConnectionString = strConn
        'Connect to the database
        .open()
    End With
End Sub
```

Add a ComboBox control to the form object and name it `cbxCubeList`. This control will hold the list of cubes present in the OLAP database. We already know how to do this, so let's go ahead and code a procedure that will do it. The following code snippet is the implementation of the `GetCubes()` procedure and the call to `Connect()` and `GetCubes()` in the form's Load event:

```

Sub GetCubes()
    'Object to hold the Catalog object
    Dim objCatalog As Object
    'Object to hold one particular cube
    Dim Cube As Object

    'Instantiates a catalog object
    objCatalog = CreateObject("ADOMD.Catalog")

    'Set up the combo box
    With cbxCubeList
        .Items.Clear()
        .DropDownStyle = ComboBoxStyle.DropDownList
        .Sorted = True
    End With

    With objCatalog
        'Connect the catalog object to the (database)
        .ActiveConnection = strConn
        'Populate the drop down
        For Each Cube In .CubeDefs
            With cbxCubeList
                .Items.Add(Cube.name)
            End With
        Next
    End With

    'Free memory
    objCatalog = Nothing
End Sub

Private Sub frmOLAPManip2_Load( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles MyBase.Load

    Connect()
    GetCubes()

End Sub

```

User interface is everything! Let's add some code to help our controls resize seamlessly:

```

Private Sub frmOLAPManip1_Resize( _
    ByVal sender As Object, _
    ByVal e As System.EventArgs) _
    Handles MyBase.Resize

    With AxCubeBrowser
        .Left = 1
        .Width = Me.ClientSize.Width - 10
        .Height = Me.ClientSize.Height -
            cbxCubeList.Height - 10
    End With
    With cbxCubeList
        .Width = Me.ClientSize.Width
    End With
End Sub

```

When the user selects a cube from the drop-down list, we want the CubeBrowser control to show information about the cube. This logic is implemented inside the cbxCubeList's SelectedIndexChanged event handler:

```

Private Sub cbxCubeList_SelectedIndexChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles cbxCubeList.SelectedIndexChanged

    With AxCubeBrowser
        'Connect the browser to the database
        'and specify the cube that you want to
        'browse.
        .Connect(objADOConn, cbxCubeList.SelectedItem)
    End With
End Sub

```

This is all that is required to browse a cube using the CubeBrowser! As you can see, not much logic or unmanaged code is required. So I recommend that you take this approach for cube browsing.

The following code snippet is the full implementation of the frmOLAPManip1 class:

```

Public Class frmOLAPManip1
    Inherits System.Windows.Forms.Form

    Dim strConn As String
    Dim objADODConn As Object

    #Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form
        'Designer.
        InitializeComponent()

        'Add any initialization after the
        'InitializeComponent() call

    End Sub

    'Form overrides dispose to clean up the component
    'list.
    Protected Overrides Sub Dispose
        (ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.IContainer

    'NOTE: The following procedure is required by the
    'Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    Friend WithEvents AxCubeBrowser As
        AxOLAPCubeBrowser.AxCubeBrowser
    Friend WithEvents cbxCubeList As
        System.Windows.Forms.ComboBox

    <System.Diagnostics.DebuggerStepThrough()> Private Sub

```

```

InitializeComponent()
    Dim resources As System.Resources.ResourceManager
        = New System.Resources.ResourceManager
            (GetType(frmOLAPManip1))
    Me.AxCubeBrowser = New
        AxOLAPCubeBrowser.AxCubeBrowser()
    Me.cbxCubeList = New
        System.Windows.Forms.ComboBox()
    CType(Me.AxCubeBrowser, System.ComponentModel.
        ISupportInitialize).BeginInit()
Me.SuspendLayout()
'
'AxCubeBrowser
'
Me.AxCubeBrowser.Enabled = True
Me.AxCubeBrowser.Location = New
    System.Drawing.Point(0, 24)
Me.AxCubeBrowser.Name = "AxCubeBrowser"
Me.AxCubeBrowser.OcxState =
    CType(resources.GetObject("AxCubeBrowser.
        OcxState"), System.Windows.Forms.AxHost.
        State)
Me.AxCubeBrowser.Size = New
    System.Drawing.Size(448, 384)
Me.AxCubeBrowser.TabIndex = 0
'
'cbxCubeList
'
Me.cbxCubeList.DropDownStyle =
    System.Windows.Forms.ComboBoxStyle.Simple
Me.cbxCubeList.Name = "cbxCubeList"
Me.cbxCubeList.Size = New System.Drawing.Size(448,
    21)
Me.cbxCubeList.TabIndex = 1
'
'frmOLAPManip1
'
Me.AutoScaleBaseSize = New System.Drawing.Size(5,
    13)
Me.ClientSize = New System.Drawing.Size(456, 411)
Me.Controls.AddRange(New System.Windows.Forms.
    Control() {Me.cbxCubeList, Me.AxCubeBrowser})
Me.Name = "frmOLAPManip1"
Me.Text = "Olap Example 1"
CType(Me.AxCubeBrowser, System.ComponentModel.

```

```

        ISupportInitialize).EndInit()
        Me.ResumeLayout(False)

    End Sub

#End Region

Private Sub frmOLAPManip2_Load( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles MyBase.Load

    Connect()
    GetCubes()
End Sub

Private Sub frmOLAPManip1_Resize( _
    ByVal sender As Object, _
    ByVal e As System.EventArgs) _
    Handles MyBase.Resize

    With AxCubeBrowser
        .Left = 1
        .Width = Me.ClientSize.Width - 10
        .Height = Me.ClientSize.Height -
            cbxCubeList.Height - 10
    End With
    With cbxCubeList
        .Width = Me.ClientSize.Width
    End With
End Sub

Sub Connect()
    'Connection string
    strConn = "Data Source= LOCALHOST; Provider=
        MSOLAP;" _
        & "Database=FoodMart 2000;UserId=" _
        & "Password="
    'Creates an ADO connection object.
    objADODConn = CreateObject("ADODB.Connection")

    With objADODConn
        'Sets the connection string
        .ConnectionString = strConn
    End With
End Sub

```

```
        'Connect to the database
        .open()
    End With
End Sub

Private Sub cbxCubeList_SelectedIndexChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles cbxCubeList.SelectedIndexChanged

    With AxCubeBrowser
        'Connect the browser to the database
        'and specify the cube that you want to
        'browse.
        .Connect(objADConn, cbxCubeList.SelectedItem)
    End With
End Sub

Sub GetCubes()
    'Object to hold the Catalog object
    Dim objCatalog As Object
    'Object to hold one particular cube
    Dim Cube As Object

    'Instantiates a catalog object
    objCatalog = CreateObject("ADOMD.Catalog")

    'Set up the combo box
    With cbxCubeList
        .Items.Clear()
        .DropDownStyle = ComboBoxStyle.DropDownList
        .Sorted = True
    End With

    With objCatalog
        'Connect the catalog object to the (database)
        .ActiveConnection = strConn
        'Populate the drop down
        For Each Cube In .CubeDefs
            With cbxCubeList
                .Items.Add(Cube.name)
            End With
        Next
    End With
End Sub
```

```
'Free memory
objCatalog = Nothing
End Sub

Protected Overrides Sub Finalize()
    objADODConn = Nothing
    MyBase.Finalize()
End Sub
End Class
```

Summary

In this chapter you read about the best practices when it comes to programming OLAP applications in the .NET Framework. It is advisable that you use wrapper classes for your existing COM and ActiveX components rather than try to embed unmanaged code into your applications. There are many indications that future versions of SQL Server will run natively on the .NET Framework, and by then, we expect to have some sort of data provider for the Analysis Services.

The Object-Oriented Features of VB .NET

In This Appendix

This appendix provides an overview of object-oriented programming and covers the features in VB .NET that allow developers to write object-oriented applications for the .NET Framework.

Topics discussed are:

- Overview of OOP support in Visual Basic in the past
- Object-oriented programming concepts
 - Classes and objects
 - Members, properties, and methods
 - Inheritance and polymorphism
- Development of a small object-oriented application

VB .NET is an Object-Oriented Language

Visual Basic .NET is an object-oriented programming language. In terms of the type of complex objects that can be built using VB .NET, the language looks nothing like what its predecessors offered to developers. The strategy behind the creation of VB .NET was “keep it simple, make it more powerful.” With this in mind,

Microsoft released a product that was as easy and intuitive to use as VB 6, yet as powerful as the Visual C++ architecture. Object-oriented programming opens a lot of doors for traditional VB programmers and takes them to a whole new world and programming paradigm.

This reality is so evident today that many C++ programmers who want to develop for the .NET Framework are moving away from Visual C++ and adopting Visual Basic .NET and Visual C# as the languages of choice.

In a mission-critical web application, VB .NET can be used to build the powerful components of the business layer. When it comes to the development of ASP .NET pages, the developer has a choice between VB .NET and C#. With these two languages, the Internet becomes an object-oriented environment powered by the .NET Framework.

OOP Support in Visual Basic 5 and 6

Microsoft started implementing limited support for object-oriented programming in Visual Basic 5. The introduction of class modules allowed developers to create classes that could be used as complex data types in Visual Basic code. This functionality made VB a better OOP language. However, the classes that one could create were very limited because they did not support the more advanced concepts of OOP, such as inheritance and polymorphism.

Shortly after the release of its Component Object Model (COM) standard, Microsoft came out with Visual Basic 6. VB 6 extended the limited object-oriented programming functionality that was present in VB 5.

Object-Oriented Programming Concepts

Before you can exploit the full power of the object-oriented features in VB .NET, it is important for you to know exactly what object-oriented programming is and understand some of its major concepts.

Classes and Objects

Classes and objects are at the core of the object-oriented programming paradigm. They form the basis of everything that goes on during program design and execution. Before you can understand the fundamental concepts of classes and objects, it is important to know a little about their roots from a computer science perspective.

In computer science, a *data type* refers to a mathematical set containing values of the same sort. A sort, in this case, is a defined number of operations that can be conducted on a particular value. For example, you can add, multiply, divide, and subtract all values of the type Integer with some exception rules. You cannot do these operations on values of type Boolean. This is because Boolean values belong to the Boolean set while the Integer values belong to the Integer set. Such basic sets of values are called *primitive data types*.

Advances in computing, pushed by pressure from business data processing, required that data types be more complex than the primitive types. This was the reason for the emergence of *composite data types*, like arrays, lists, and records.

Programming language linguistics went further. People started to think about the feasibility of having one data type that could incorporate a set of things—tangible entities, not just simple values or lists of values.

Object-oriented programming, the idea of classes and objects, grew out of this thought. The definitions for object and classes are two simple phrases:

- An object is a programmer-defined data type.
- A class is a module of code that defines an object.

The OOP paradigm provides the developer with a rich set of tools and standards that can be used to manipulate and work with objects more easily. Like all entities around you, objects can have attributes and behaviors. The most interesting of all is that you, the programmer, define these attributes and behaviors through classes. They can be as complex, as clever, or as obtuse as you want them to be.

Implementing Classes in VB .NET

Everything in VB .NET is an object. Whether it is a form or a control, the VB .NET entity provides the developer with a rich set of attributes, methods, and events to work with.

In the next few sections, we will walk through the development and consumption of a simple class using VB .NET. The class does not have a commercial purpose; it is simply to teach you the practice of OOP in VB .NET. You will notice that the class expands as we go further into the chapter. This is an attempt to get you up and running quickly in your ultimate goal to develop classes. We cover the concepts of OOP by example. The project that is used for this example is the AppendixOne.sln solution provided on the companion CD-ROM with this book.

Assuming that you have correctly installed Visual Basic .NET, load the development environment. The view should be similar to that illustrated in Figure A-1.

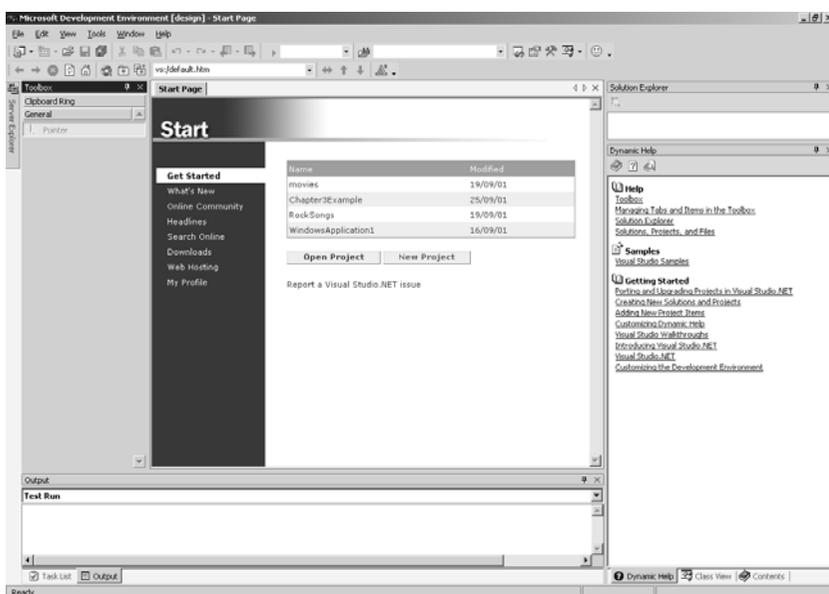


Figure A-1: The Visual Studio .NET development environment

Planning Ahead

The project you are about to create does something quite simple. It loads a form with two input controls into which you enter some information about your favorite customer. Then, it creates a Customer object and assigns these attributes to it.

To achieve this amazing feat, you are going to work with two main objects, a visual form and a non-visual class. The forms you program with in VB .NET are also objects instantiated from classes, but these classes are built into the .NET Framework Common Language Runtime to facilitate your job as a web developer. There are other classes in the CLR that you will learn about later in this book.

The Customer class is the primary focus of the following section because you are going to walk through its design

and implementation while learning the ropes of object-oriented programming.

As we go along in this appendix, you will be able to see how objects are created and used in VB .NET.

Creating a Class

Before we go into creating the form, which is a quite simple task, let's take a good look at the creation of a class. First, create a new Visual Basic .NET Windows Application project. Save it to a convenient location and name it AppendixOneExample.

Setting Up the GUI

In this section, you will prepare the user interface that will be used with the Customer object. This is the only means through which the user can gain access to the object and its attributes.

1. The new solution project loads a form by default. Change the form's name to **frmExample**. Click on the form to give it focus.
2. From the Windows Forms tab of the toolbox pane, drop a Label control onto the form. Name it **lblTitle**, align it properly, and set its Text property to **Title**.
3. From the Windows Forms tab of the toolbox pane, drop another Label control onto the form. Name it **lblFirstName**, align it properly, and set its Text property to **First Name**.
4. To the left of the newly added control, drag and drop a TextBox control onto the frmExample. Name it **txtFirstName** and empty its Text property. This is used as the input control for the customer's first name.

5. Copy and paste `lblFirstName` and `txtFirstName` onto the window. Align the pasted controls just below the previous ones. Rename the label to **`lblLastName`** and the textbox to **`txtLastName`**.
6. Paste the two controls onto the form again. This time, label the two controls **`lblAddress`** and **`txtAddress`**, respectively. Set the text property of `lblAddress` to **`Address`**. `txtAddress` is the input control for the customer's address. Set its `MultiLine` property to **`True`**.
7. Below `txtAddress`, add a command button. Name it **`cmdInstantiate`**, and label it **`Instantiate Customer`**. This is the button that will trigger the creation of a `Customer` object with the input values on `frmExample`.

The `frmExample` form should look similar to the illustration in Figure A-2. It is a good idea to play around with some of the properties that the form object has in VB .NET. This is not our focus here, so let's move on.

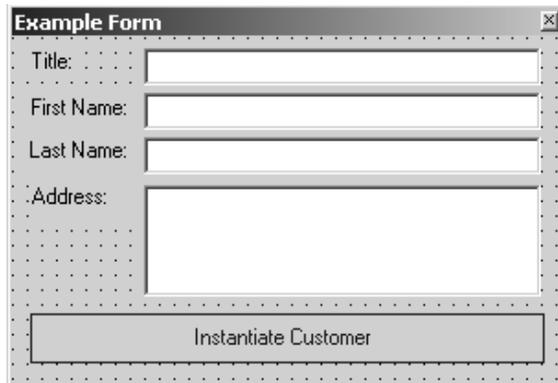


Figure A-2: The `frmExample` form

Loading a New Empty Class

The user interface that instantiates and consumes the object is ready. In this section, you will create a new, empty class and add it to the project. If you are accustomed to creating class modules in Visual Basic 6, this task is not new to you. Follow these steps to create and add a new class to AppendixOneExample:

1. From the File menu, click the **Add New Item** option.
2. In the Add New Item dialog, illustrated in Figure A-3, you are given a choice to select the type of new item that you want to create. In the Categories tree list, select the **Local Project Items** node. The items for that particular category are listed in the Templates list on the right. Click on the **Class** item.

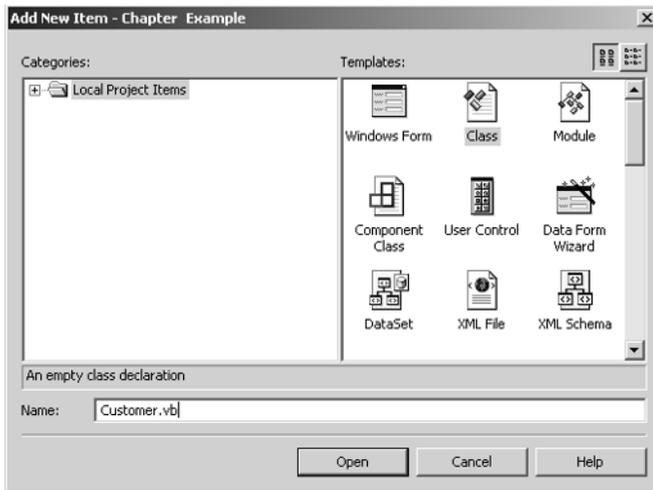


Figure A-3: The Add New Item dialog

3. In the Name field, specify the name for the new class. Call it **Customer.vb**.

4. Click the **Open** button to load the new class as part of the project. The code window for the class opens with the following code:

```
Public Class Customer
End Class
```

Public Class Customer is the statement that declares the Customer class. The End Class statement is the last statement in the chunk of code that is part of the class. Anything outside these two statements is not part of the class. As illustrated in Figure A-4, the Solution Explorer shows the newly added Customer class.

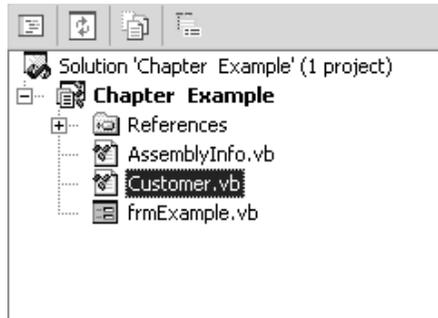


Figure A-4: The Solution Explorer showing the Customer.vb class

The class is expanded as object-oriented programming concepts are introduced in the subsequent pages.

Properties and Members

Properties and members are the two most important characteristics that an object has. You already learned that an object is a complex data type made of simpler data types or even more complex objects. Properties and members define the amount and type of values that an object can hold.

A member is often referred to as a *member variable*. It is a variable of any type that is initialized as part of the object when it is instantiated. An object's member variables have object scope only; they cannot be used nor referenced, and do not exist outside the object's definition, which is the class. An example of a member variable is a string containing the first name of a customer. Member variables are used for the internal process of the object. Its content is usually hidden by a technique called encapsulation, which is covered later in this appendix.

A *property* is an attribute that an object has. Similar to a member, a property can hold different values of the same type. However, in a properly planned object-oriented application, properties are used to encapsulate—that is, to hide a member's value. At other times, they provide a better visual presentation of an object's data to a user. This might be the case when a user reads a property called name, which is, in fact, the concatenated result of the firstname and lastname member variables of an object.

A user usually sets attributes for an object. It is, therefore, fair to draw a conclusive argument on the relationship of properties and members. The members are the data values that the object contains, while the properties are the information obtained from these values. This relationship is illustrated in Figure A-5. As can be perceived, the properties form an interface between any calling application and an object. Values for the internal processes of the object are stored in its member variables. Internal operations are carried out by member functions, while methods are also part of the interface. Methods are covered later.

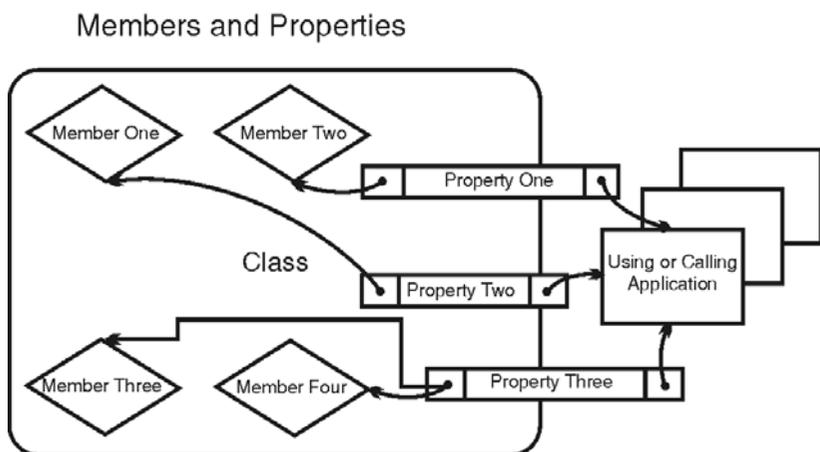


Figure A-5: The relationship between member variables and properties

Implementing Members and Properties for the Customer Object

Now that the fundamental and conceptual bases of members and properties have been established, you are ready to implement members and properties for the class that serves as the code base for instantiating Customer objects.

Once instantiated, the Customer object will be able to perform the following tasks:

- Allow users to assign values to its attributes. The following attributes must be available:
 - **title**: Title of the customer (i.e., Dr. for Doctor)
 - **firstName**: First name of the customer (i.e., Terrence)
 - **lastName**: Last name of the customer (i.e., Joubert)
 - **address**: Address of the customer (i.e., 78 Drake Street, MA 02120)

- **salutation:** The full name and title for the customer (i.e., Dr. Terrence Joubert)
- Provide a generic structure and architecture from which future objects can inherit

Members

Adding members to a VB .NET class is as simple as declaring variables in any module of Visual Basic code. The syntax for declaring a member variable is:

```
access_specifier variable_name [As DataType]
```

`access_specifier` provides instructions to the compiler about the level of access that the member variable should have. Note that the `access_specifier` keywords that are described here apply to all types of entities that one can create inside a class. The keywords are three possible access levels for a class member variable.

- **Private:** This keyword denotes that the member variable is available to the code in that particular class only. Functions and procedures inside the class can reference this variable. Nowhere else outside the class or inside its inherited children will you be able to reference a private member variable.
- **Protected:** This keyword denotes that the member variable is available to the class in which it is declared and all the classes that inherit from it. Functions and procedures inside the class `Dog` inherited from the class `Mammal` are able to reference a `Mammal` member variable of access specifier `Protected`. Inheritance is covered later in this chapter.
- **Public:** This keyword denotes that the member variable can be referenced anywhere in the program. It acts like a property, in the sense that its value can be read and written to like that of any normal variable. In well-written object-oriented applications, the

Public access specifier is not used on member variables of classes. Doing so will violate the rules of encapsulation of object data. After instantiating a Dog object inside a code module, it is possible to make it bark by calling its Bark() method of Public access specifier, but it must be impossible to change the dog's voice, because then it will lose its uniqueness and personality.

variable_name is the identifier of the member variable. It must follow the same legal rules for all identifiers in the Visual Basic language syntax.

The [As DataType] clause is the data type of the member variable. It is optional to specify a data type for a member. Not doing so will cause VB .NET to mark the variable as having the type System.Object, and doing so will severely affect the performance of the object instantiated from the class. For this reason, I strongly recommend specifying a defined data type and leave System.Object for variables that are not predetermined.

Customer Members

From the list of attributes that we defined for the customer earlier, we now have a very clear idea of what members the customer will have.

Open the code module for the Customer.vb class. Just after the Public Class Customer statement, type the following lines of code:

```
Private strTitle As String
Private strFirstName As String
Private strLastName As String
Private strAddress As String
```

This declares four member variables for the Customer class. The names of the variables are quite self-descriptive in relation to their purposes.

Properties

The properties that a Customer class has are the same attributes that were outlined above:

- title
- firstName
- lastName
- address
- salutation

The first step to adding a property to a class is to determine what information the property will give to an end user or developer who happens to be working with the object. Then comes the issue of how the property should be used. A property may be read-only or allow read and write. The details about the properties of the Customer class are listed in Table A-1.

Table A-1: The properties and their content

Property Name	Purpose	Content	Usage
title	Title of the customer	strTitle member variable	Read and write
firstName	First name of the customer	strFirstName member variable	Read and write
lastName	Last name of the customer	strLastName member variable	Read and write
address	Address of the customer	strAddress member variable	Read and write
salutation	Salutation for the customer	strTitle + <space> + strFirstName + <space> + strLastName	Read-only

Implementing Properties

A property is implemented using a type of procedure called the *Property procedure*. Property procedures are

special because they hold two other procedures inside their code body. These are the Get and Set procedures. To better understand the functionality of Property procedures, it is important to understand their syntax.

Property Procedures

```
<access_modifier> Property <property_name>() As
    <data_type_x>
    Get
        'Code module for Get
    End Get

    Set (ByVal <identifier> As < data_type_x >)
        'Code module for Set
    End Set
End Property
```

The syntax for the Property procedure is very interesting. It is important that you understand each part of it.

```
<access_modifier> Property <property_name>() As
    <data_type_x>
```

This part of the syntax is the declaration of the procedure. Notice the End Property statement that terminates the code body of the procedure. As you have noticed, the Property procedure is declared similar to other procedures that you are accustomed to. The only exception is the use of the property statement instead of procedure. The <property_name> clause defines a name for the property. This name must conform to the legal identifier names allowed in Visual Basic .NET.

<data_type_x> defines a data type for the particular property. This can be of any legal data type.

Get Procedure

```
Get
    'Code module for Get
End Get
```

Get is a special procedure that returns the value of the property. The Get procedure is automatically executed when the calling code attempts to retrieve a property value with the statement:

```
Object.Property
```

Any logic can be coded inside the Get procedure, given of course that the logic makes sense in the context of retrieving a property value. Anything that happens whenever the calling code retrieves the value of a property must be coded inside the Get procedure.

Set Procedure

```
Set (ByVal <identifier> As < data_type_x >)  
    'Code module for Set  
End Set
```

The Set procedure is called whenever the calling code attempts to assign a value to a property with this statement:

```
Object.property = anyValue
```

<identifier> defines any valid identifier that you may specify. This identifier references the value that is assigned to the property. In the case of the calling statement above, <identifier> would reference anyValue. The two should be of the same data type specified in < data_type_x >.

As with the Get procedure, the code body of the Set procedure can hold any form of logic. The most relevant however, is validating the value before assigning it to the property.

The Set procedure also plays a key role in the case of a read-only property. To specify a property as read-only, do not perform an assignment inside the Set procedure. To create a better user interaction, you may opt to pop up a message that tells the user or developer that a value

cannot be assigned to the property because it is read-only. If you do not wish to have a user-friendly Set property procedure, use the access modifier `ReadOnly` for the property and do not specify a Set procedure.

Applying the Concepts

Open the code window for the Customer class. Just after the declaration of the member variables, type the following code to define the Title, FirstName, LastName, and Address properties for the Customer class:

```
'Begin Properties
'Property Procedure for the Title property
Public Property Title() As String
    Get
        Return strTitle
    End Get

    Set(ByVal Value As String)
        strTitle = Value
    End Set
End Property
.....

'Property Procedure for the FirstName property
Public Property FirstName() As String
    Get
        Return strFirstName
    End Get

    Set(ByVal Value As String)
        strFirstName = Value
    End Set
End Property
.....

'Property Procedure for the LastName property
Public Property LastName() As String
    Get
        Return strLastName
    End Get
```

```

        Set(ByVal Value As String)
            strLastName = Value
        End Set
    End Property
    .....

'Property Procedure for the Address property
Public Property Address() As String
    Get
        Return strAddress
    End Get

    Set(ByVal Value As String)
        strAddress = Value
    End Set
End Property
.....
'End Properties

```

The Get and Set code is very straightforward. When the calling code retrieves the value of the property, the Get procedure returns the value in the corresponding member variable. When the calling code assigns a value to the property, the Set procedure assigns the value of the assigned value of the corresponding member variable. This method of hiding the value of the member variables is called *encapsulation*. It ensures that the data inside the object is secure at all times. For example, using Property procedures, you can perform a complex security check on the current user of the application before allowing this individual to either retrieve or assign a value to a class member.

The Property procedure for the Salutation property has a little more overhead. Salutation is read-only, meaning that you cannot assign it a value. Another issue is that it is a computed property. It returns the value of more than one member variable. Write the code body for the Salutation Property procedure as the following:

```
'Property Procedure for the Salutation property
ReadOnly Property Salutation() As String
    Get
        With Me
            'compute the salutation and return it
            Return .Title + ". " + .FirstName + " "
                + .LastName
        End With
    End Get
End Property
.....
```

This code marks the property as read-only and returns the salutation of the customer. A typical value for the Salutation property would be Mr. John Doe.

Testing the Results

In this section, we instantiate a Customer object and use the controls on frmExample to input values into properties and see how it functions.

1. Open the frmExample Windows Form object.
2. Double-click the cmdInstantiate command button control to get to its click() event handler. This event is triggered when the button is clicked. Add the following code to the event handler:

```
Dim cust As New Customer()
With cust
    .Title = Me.txtTitle.Text
    .FirstName = Me.txtFirstName.Text
    .LastName = Me.txtLastName.Text()
    .Address = Me.txtAddress.Text
    MsgBox(.Salutation & vbCr & " Lives at: " &
        vbCr & .Address)
End With
cust = Nothing
```

When the button is clicked, a new Customer object is instantiated and its properties are assigned the values from the input controls on the window. A message box appears, telling you the salutation and address of the

customer. Then the Customer object is released from memory.

3. Run the example, input some information in the controls, and click the button. The result should be something like that illustrated in Figure A-6.

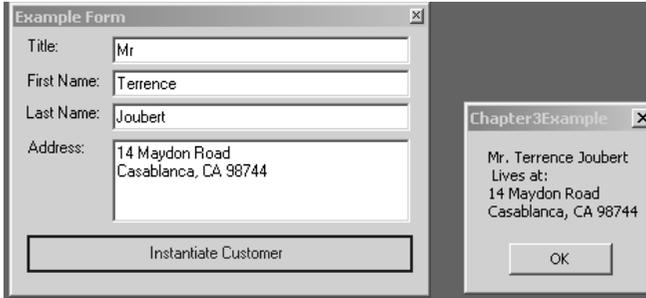


Figure A-6: The result of the sample application

Methods

Methods are the elements of a class that implement the particular behaviors of the resulting object. At first sight, methods may seem like simple functions inside the code module of a class. In fact, they are, but by the strict rules of OOP, every method must provide one specific behavior to an object. A behavior in this case is an action that an object can take or a reaction that is a response to the surrounding environment. Therefore, methods must be carefully planned and their actions properly coordinated.

Inside a class, a method is implemented similar to a modular regular or global function. In fact, a method can take the form of a function or a procedure, depending on what you intend to achieve. If you hope to return a value, use functions; otherwise, a procedure approach will do the job.

Inheritance

The principle of inheritance is that a class can be derived from another class. This relationship results in objects that inherit the properties, members, and methods of other objects. A good example is to think of the class Dog as one that is derived from the class Mammal. A Dog object would behave similar to a Mammal and, in fact, have all the attributes of the Mammal, but its properties and methods are somewhat specific to “doggy” behavior. In a scenario of developing an application that incorporates a lot of related classes (Dolphin, Human, Ape, Tiger), such a principle allows the developer to code a generic class—Mammal, from which the other classes can inherit behaviors and properties. The chain of inheritance can go further to include the classes illustrated in Figure A-7:

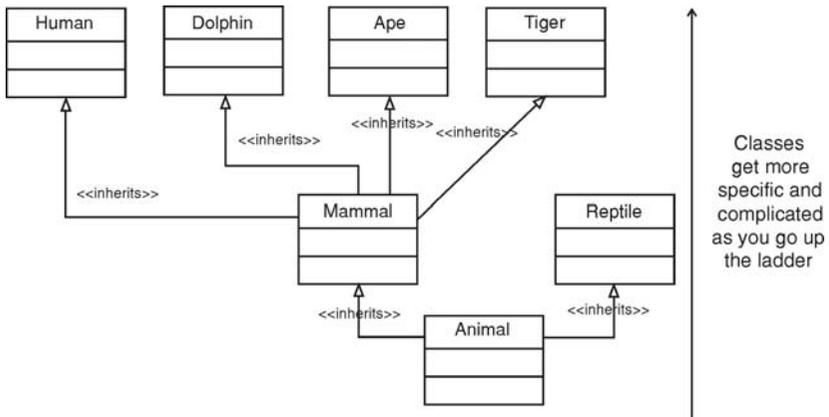


Figure A-7: A chain of inheritance

Unlike its predecessors, the VB .NET object-oriented architecture supports the full power of inheritance, like C++, C#, and Java. This has helped it become the favorite .NET language among developers who prefer

development speed and ease of use over the classical taste of the other languages.

In VB .NET, the following rules apply to inheritance:

- A class can be optionally marked as non-inheritable, meaning that no other class can be derived from it. This is done using the `NotInheritable` keyword to declare the class:

```
<access_modifier> NotInheritable Class <classname>  
    'Class code body  
End Class
```

- A derived class can implement a method that overrides or extends the same functionality of the same method of its parent class.
- A class can be derived from one class only. There is no support for multiple inheritance.
- A class can also be declared as `MustInherit`. In this case, your code cannot consume its instance directly but rather consumes the instance of a class derived from it.

Implementing Inheritance in VB .NET

The syntax for implementing an inherited class is:

```
Public Class ClassX  
    Inherits ClassY  
    ' Overrides, overloads, and extends members  
    ' inherited from the base class  
End Class
```

Notice the use of the `Inherits` keyword after the class declaration. This tells VB .NET that `ClassX` is derived from `ClassY`. VB .NET does not support multiple inheritance, so it is impossible to have two `Inherits` statements.

After the base class is specified, you have the option to override or overload methods and extend the base class members.

Inheritance at Work

In this section, we attempt to extend the AppendixOne-Example solution to demonstrate the principle of inheritance at work. Load the solution into the development environment.

The Problem and the Solution

Upon testing the current functionality of the application, you have found that one thing is left out. For later analysis purposes, you want to obtain the customer's zip code and state of residence. Since you hope to market the application internationally, requiring a zip code and state in the current customer functionality will cause trouble with international customers. You decide to create a new `USCustomer` class that inherits generic customer attributes and behaviors, while providing an extended functionality for American customers.

1. The first thing you need to do is change two things with the `Customer` class. Change the declaration of member variable `strAddress`, setting its access modifier to **Protected**. This will allow the code in any inherited class to reference this variable directly. Then, place the keyword **Overridable** between the keywords `Public` and `Property` in the declaration section of the `Address` property. It should now look like this:

```
Protected strAddress As String
    'Other parts of the class here
Public Overridable Property Address() As String
    Get
        Return strAddress
    End Get
    Set(ByVal Value As String)
```

```

        strAddress = Value
    End Set
End Property

```

2. Create a new class and name it **USCustomer**. Then enter the following code:

```

Public Class USCustomer
    Inherits Customer

    Private strZipCode As String
    Private strState As String

    Public Property zipCode() As String
        Get
            Return strZipCode
        End Get

        Set(ByVal Value As String)
            strZipCode = Value
        End Set
    End Property

    Public Property state() As String
        Get
            Return strState
        End Get

        Set(ByVal Value As String)
            strState = Value
        End Set
    End Property

    'The new address property overrides the one in
    'Customer
    Public Overrides Property Address() As String
        Get
            Return strAddress & " " & strState & " " &
                strZipCode
        End Get
        Set(ByVal Value As String)
            strAddress = Value
        End Set
    End Property
End Class

```

There are a couple of interesting things that are happening in this class. First, we declare two member variables to hold values for the zip code and state, respectively. Then, the zipCode and State properties are added. The most interesting inheritance feature in this class is overriding the Address property of the base Customer class. The keyword Overrides is used to perform this task. In order to use this keyword on an entity in a derived class, the keyword Overridable must be specified on the declaration of that particular entity in the base class. Overriding methods and member variables work the same way as overriding properties in Visual Basic .NET. This allows you to call one method that may perform different actions, depending on which type of object you are calling it on. This feature is called *polymorphism* in object-oriented programming.

Now you are ready to test the new USCustomer class.

1. Open the frmExample form object.
2. Add two pairs of label and textbox controls to the form. Name the labels **lblZipCode** and **lblState** and set their Text property accordingly. Name the text boxes **txtZipCode** and **txtState**.
3. Open the code body of the click() event handler for the cmdInstantiate command button. Replace the code in that handler with the following:

```
Private Sub cmdInstantiate_Click(ByVal sender As
System.Object, _
ByVal e As System.EventArgs) Handles
cmdInstantiate.Click

    Dim USCust As New USCustomer()

    With USCust

        .Title = Me.txtTitle.Text
        .FirstName = Me.txtFirstName.Text
        .LastName = Me.txtLastName.Text
        .Address = Me.txtAddress.Text
    End With
End Sub
```

```
.zipCode = Me.txtZipCode.Text
.state = Me.txtState.Text
MsgBox(.Salutation & vbCr & " Lives at: " &
vbCr & .Address)

End With

USCust = Nothing

End Sub
```

The code does only a few new things: instantiates a new `USCustomer` object and assigns the appropriate values to its `State` and `zipCode` properties.

Summary

In this appendix, you have explored the features of Visual Basic .NET that make it an object-oriented programming language. VB .NET, an OOP language, is as powerful as Visual C++ and Visual C#.

Database Normalization

In This Appendix

This appendix provides a definition of normalization and explains the process of normalization.

Normalization

Normalization refers to a series of steps taken to ensure the most efficient relational structure of a database. The process usually involves the division of one existing database table into two or more tables and maintaining the proper relationship patterns between them. Therefore, a modification to one field in a table that is part of a normalized set of tables propagates throughout the other tables in the set using the defined relationships.

► **Note:** Notice the use of the word “propagate.” It does not mean that the data in the modified field physically moves to any other field. It simply means that the other tables easily reference the data because it is related to them.

The Normalization Process

The formal process of normalization centers around the concepts of some defined normal forms that the database designer can use to achieve a good relational structure.

The normal forms define the logical steps that are used to normalize database tables. In the following sections, the concepts defined by each normal form are covered along with a good example of the normalization process.

The concepts are defined and taught in the context of using a real-world example. The example is of no commercial value; its purpose is for the teaching of a conceptual topic.

Here is the scenario. We have a database table that stores data about orders made by customers. This table is illustrated in Figure B-1.

Order	AccountNumber	Customer	Address	ProductDescription	Quantity	UnitPrice	ItemTotal	OrderTotal	DateOrdered
2674	ALI-096864445	Jane Dickerson	86 Blue Hill Avenue	Guide to E-Commerce	2	12.34	24.68	434.25	30/07/01
2674	ALI-096864445	Jane Dickerson	86 Blue Hill Avenue	Batch File Programming	1	24.99	24.99	434.25	30/07/01
2674	ALI-096864445	Jane Dickerson	86 Blue Hill Avenue	Algorithm Design	4	54.67	218.68	434.25	30/07/01
2674	ALI-096864445	Jane Dickerson	86 Blue Hill Avenue	Database Concepts	7	23.7	165.9	434.25	30/07/01
4563	LOP - 3947845	Vicky Haynes	45 Wood Street	Web Design Concepts	3	43.8	131.4	389.6	30/07/01
4563	LOP - 3947845	Vicky Haynes	45 Wood Street	Linux Security	9	12.98	116.82	389.6	30/07/01
4563	LOP - 3947845	Vicky Haynes	45 Wood Street	C++ Programming	3	23.8	71.4	389.6	30/07/01
4563	LOP - 3947845	Vicky Haynes	45 Wood Street	Java Beans	2	34.99	69.98	389.6	30/07/01

Figure B-1: Customer orders data

The application software that uses this table for read and write purposes has been experiencing performance troubles lately. This table requires normalization. We will use it as an example to illustrate the process of normalization.

The First Normal Form

The First Normal Form is the first step to normalizing a table. When a table has undergone this step, it is referred to as being in the First Normal Form, or 1NF. From looking at the layout of the table in Figure B-1, three things are easily noticeable: repeating groups, object inconsistency, and storage space.

Repeating Groups

The table is repeating many of the same values. For every order detail, the user must enter the information for the order and the customer. For an order with 20 or more items, this is a daunting task.

Furthermore, there is the risk of a user making a mistake in typing the product name into the ProductDescription field. If you retrieve a report on that particular product, you will not be presenting the right information because your filter for the query hinges on the ProductDescription text field. If the query filters retrieved data for Linux as the ProductDescription, records with Linx are filtered out of the retrieved rowset.

Object Inconsistency

As a solution for repeating groups, you may decide to remove the order-centric data from every record and put in into the first record that contains orderItem-centric data. See Figure B-2 for an illustration of this scenario.

The fact is that the table in Figure B-2 does not fit into the relational database design concepts. It is impossible to express a query that retrieves all order items for one particular order with this data entry model.

Order	AccountNumber	Customer	Address	OrderTotal	DateOrdered	ProductDescription	Quantity	UnitPrice	ItemTotal
2674	AUI-096864445	Jane Dickerson	86 Blue Hill Avenue	434.25	30/07/01	Guide to E-Commerce	2	12.34	24.68
						Batch File Programming	1	24.99	24.99
						Algorithm Design	4	54.67	218.68
						Database Concepts	7	23.7	165.9
4563	LOP - 3947845	Vicky Haynes	45 Wood Street	389.6	30/07/01	Web Design Concepts	3	43.8	131.4
						Linux Security	9	12.98	116.82
						C++ Programming	3	23.8	71.4
						Java Beans	2	34.99	69.98

Figure B-2: Removal of repetitive groups

Storage Space

The repeating values force the physical database file to grow at a faster rate. The performance of a database table hinges on the number of rows it contains. If the Sales Department is entering around 50 orders per day into this table, write performance decreases drastically because for every order item entered, customer information and order information has to be entered.

The Solution

The First Normal Form simply attempts to:

- Eliminate repeating groups in individual tables
- Create a separate table for each entity type
- Identify each row with a primary key

A good look at the table in Figure B-1 reveals that the order details form a repeating group. Our solution to this problem is to split this table into two separate tables, one storing order details and the other storing the orders, as illustrated in Figure B-3.

The Order Table

Order	AccountNumber	Customer	Address	OrderTotal	DateOrdered
2874	AUI-096864445	Jane Dickerson	86 Blue Hill Avenue	434.25	31/07/01
4563	LOP - 3947845	Vicky Haynes	45 Wood Street	389.6	31/07/01

The OrderDetail Table

Order	Product	ProductDescription	Quantity	UnitPrice	ItemTotal
2874	54	Guide to E-Commerce	2	12.34	24.68
2874	67	Batch File Programming	1	24.99	24.99
2874	89	Algorithm Design	4	54.67	218.68
2874	765	Database Concepts	7	23.7	165.9
4563	43	Web Design Concepts	3	43.8	131.4
4563	98	Linux Security	9	12.98	116.82
4563	56	C++ Programming	3	23.8	71.4

Figure B-3: Splitting the table into two separate tables

The two are related by a many-to-one foreign key from the OrderDetail table to the Order table, as shown in Figure B-4.

Order	Product	ProductDescription	Quantity	UnitPrice	ItemTotal
AccountNumber					
Customer					
Address					
OrderTotal					
DateOrdered					

Figure B-4: Relating the Order and OrderDetail tables

Each order detail has a composite primary key of its order and product numbers. A single entity is easily retrieved from these tables, as repeating groups, inconsistency, and storage space problems are eliminated.

These tables are in 1NF, and they are not at all perfect. The Second Normal Form identifies and resolves further issues with these tables.

The Second Normal Form

The Second Normal Form is the second step to normalizing a database table. After undergoing this step of the normalization process, the table is said to be in 2NF.

In this section, we continue to normalize the tables in our 1NF example.

Functional Dependency

The functional dependency of a table column defines whether the column is dependent on another column of the same table or some other table.

The Second Normal Form attempts to remove columns that are functionally dependent on any part of a composite key.

After taking a close look at the OrderDetail table, notice that each of the products has one description and one price. Price and description are therefore functionally dependent on the product. To enter a new product, you have to enter an order detail. This is inefficient in the business world. The table has to be in 2NF, and since the product is part of a composite key, this occurrence violates the core rule of the Second Normal Form.

The Solution

Notice that the products can be grouped into one entity type altogether. This gives your users more flexibility in entering new products as they come in while entering order details when orders are made.

The solution that puts our set of tables into 2NF is splitting the OrderDetail table into two: OrderDetail and Product, as illustrated in Figure B-5.

OrderDetail				Product		
Order	Product	Quantity	ItemTotal	Product	ProductDescription	UnitPrice
2874	54	2	24.68	54	Guide to E-Commerce	12.34
2874	67	1	24.99	67	Batch File Programming	24.99
2874	89	4	218.68	89	Algorithm Design	54.67
2874	765	7	165.9	765	Database Concepts	23.7
4563	43	3	131.4	43	Web Design Concepts	43.8
4563	98	9	116.82	98	Linux Security	12.98
4563	56	3	71.4	56	C++ Programming	23.8

Figure B-5: The OrderDetail and Product tables, which are in 2NF

The Third Normal Form

The Third Normal Form is the third step in the normalization process. It highly resembles the Second Normal Form in its attempt to remove functional dependencies. However, the Third Normal Form requires you to remove dependencies between non-key columns.

The Order table is in 2NF, but it is still vulnerable to attack from data duplication. Every time an order is made, a new customer name and account number must also be entered. Consider the situation where the Sales Department is entering 12 orders for a particular customer every week. Not surprisingly, the salespeople end up entering the customer name and account number 48 times per month. After a few years, this unnecessary duplication of data deals a severe blow to the performance of the Order table because it will be big and slow, and you will be guilty of violating the design concepts of relational database design.

Customers and their accounts can be classified into a new group—Account. In fact, when developing in the real world, you would not want a customer to possess more than one account. For our example, the division illustrated in Figure B-6 will do.

Order				Account		
Order	AccountNumber	OrderTotal	DateOrdered	AccountNumber	Customer	Address
2874	AUI-096864445	434.25	31/07/01	AUI-096864445	Jane Dickerson	86 Blue Hill Avenue
4563	LOP - 3947845	389.6	31/07/01	LOP - 3947845	Vicky Haynes	45 Wood Street

Figure B-6: The new Account group

The database tables are now in 3NF and very optimized compared to the single Order table that we started with.

How Far to Normalize

There are other forms of normalization that exist beyond the Third Normal Form. These are not used extensively in the database engineering industry anymore, so they are not covered in this appendix.

If you carefully follow the rules of the first three normal forms, you will have a good relational database design.

Normalization Concerns

Normalization is a science of compromises. Although it greatly increases the performance margin of relational databases, extensive normalization that results in several related tables can become a big performance problem and a victim of its own normalization success. Normalization is an issue that must be balanced.

Consider our example again. The query retrieving every available detail for order number 2874 is:

```
SELECT *
FROM dbo.Order
WHERE Order = 2874
```

This is straightforward. After we've put the database into 3NF, the query to achieve this same result is:

```
SELECT dbo.Order.Order, dbo.OrderDetail.Quantity,
       dbo.OrderDetail.Total, dbo.Product.ProductDescription,
       dbo.Account.Customer
```

```
FROM dbo.Order INNER JOIN
    dbo.OrderDetail ON
    dbo.Order.Order = dbo.OrderDetail.Order INNER JOIN
    dbo.Product ON
    dbo.OrderDetail.Product = dbo.Product.Product
    INNER JOIN
    dbo.Account ON
    dbo.Order.AccountNumber = dbo.Account.AccountNumber
WHERE (dbo.Order.Order = 2874)
```

The query requires us to construct complex joins in order to achieve the same result. Joins are more efficient when they are kept simple. However, there are many things that can form part of an order in the real world. Consider issues like shipment method and customer contact information that are tied to an order. Retrieving all this data using joins causes performance issues.

When normalizing, always maintain a plan as to how the normalized table is going to be used and the depth of related tables that you want to break it into.

Summary

In this appendix, we demystified the concepts of normalization, the process of optimizing database tables to avoid redundancy and waste of space that may affect read and write procedures and result in bad performance.

The normalization process has three main forms:

- The First Normal Form eliminates repetition and attempts to create a table for each entity type.
- The Second Normal Form removes the functional dependency of any column on part of a composite primary key.
- The Third Normal Form removes the functional dependency of any column on any other non-key column.

Views, Stored Procedures, and Triggers

In This Appendix

This appendix provides a discussion of views, stored procedures, and triggers, and the steps involved in their implementation inside SQL Server 2000. The main tools used in this chapter are the Query Analyzer and the Enterprise Manager.

Topics discussed are:

- The concepts of views, stored procedures, and triggers
- The usage of views, stored procedures, and triggers
- The implementation of views, stored procedures, and triggers using the Query Analyzer

Views

There are times in application programming when the developer inherits a database schema that will not change radically throughout the lifetime of the database. SQL views allows the developer to implement new schema without having to change the underlying structure of the database. In effect, it allows the developer and user to view the data from a different perspective.

Introduction to Views

A *view*, as its name suggests, is a way of presenting part of the database to a user that fits his or her needs and requirements. A view includes data derived from selected columns and rows from one or more database tables. You can think of a view as a “virtual table,” but no data is actually stored in this virtual table. The content of a view is defined by a query. The set of data that the view represents is reconstituted by the DBMS (database management system) each time a query refers to the view.

Views in Action

Views provide developers with various solutions to some common application development and data management problems. In this section we will go over some common uses of views.

Protecting the Data

As with normal tables, security restrictions can also be applied to views (virtual tables). Views can be used to restrict access to the database that different classes of users have. Groups of users can be granted access to only those views that reflect their processing needs. Using views this way, access can be restricted both vertically and horizontally—that is to say, restrictions on

columns and rows. That way, the confidentiality of certain data can be protected.

► **Note:** Permissions on both tables and views are granted for the whole object. Restrictions cannot be applied per row, only per column. However, the view definition itself can restrict which columns and rows are returned as the result set.

Let's consider a payroll system as a simple example. The administrator can create views for all users so that they can see only the employee details, such as Name, Surname, Jobtitle, and DepartmentID, but without confidential details, such as Salary (restrictions on columns). Another view could also be created for a particular user that shows all the details but only for that user (restrictions on rows). Using views that way also insulates the tables from unauthorized modification.

Simplifying Complex Queries

Views can also help SQL programmers to break down a complex query into sets of smaller, simpler queries. Each view would represent a step closer to what the programmer actually wants. The simpler views can be used in subsequent views and SELECT statements to get a step closer to the desired result.

Data Presentation

Views can also be used to represent information in a way that is more meaningful to the user. Columns can be combined and data can be formatted using various T-SQL functions, such as cast and convert. For example, a view can be used to combine Name and Surname columns into a single Fullname column. A Datetime column can be converted to a string or formatted to display long date format to avoid ambiguity.

► **Note:** Long date format displays the full name of the month and also a four-digit year. For example, 1/2/01 becomes January 02, 2001, assuming the standard we are using is U.S. If we were using the British standard, the date would be February 01, 2001.

Views can also be used to denormalize the data before presenting it to the user. Normalized tables, though efficient for data storage, are not very intuitive to humans. By using views, data can be presented in a more intuitive way from multiple tables. Instead of showing foreign keys, the value of the referenced table is shown to the user. For example, in the Northwind database that we used throughout this book, instead of showing ProductID from the Order Details table, ProductName from the Product table would be shown instead. You can also use views to present summary data to the user by using aggregate functions and GROUP BY, CUBE, and ROLLUP.

Managing Views

There are several ways that views can be implemented in SQL Server. With experience, you can learn to build views using different techniques according to the situation you are in. The main tool for building views is the New View tool in Enterprise Manager. The New View window can be accessed by right-clicking on the View node in the console tree and selecting New View from the pop-up menu.

From within the tool, you have the option of designing the view graphically from the diagram pane and further refining it in the grid pane, or you can type the desired query in the *SQL pane*. Changes in any of the three are reflected in the other two. You will find that a combination of the three would best suit your needs.

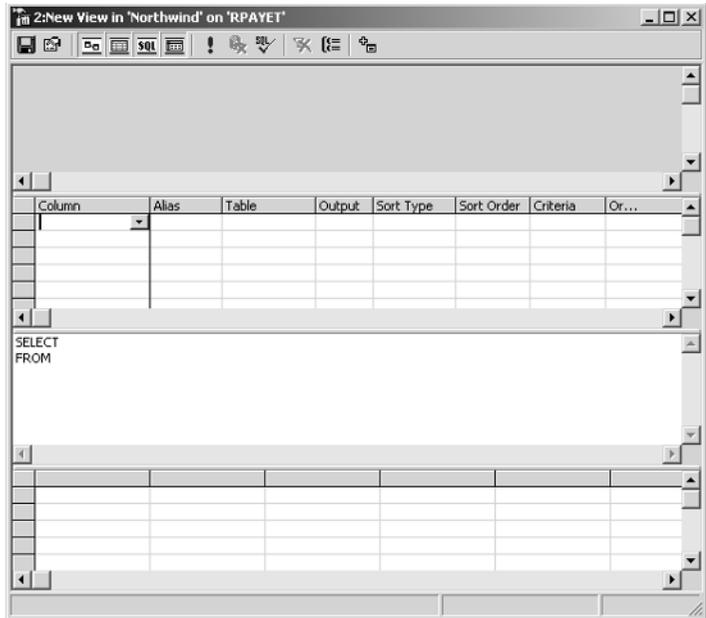


Figure C-1: The New View window



Tip: You can also use the New View window to generate template queries that you can use elsewhere in your application. Use the Diagram pane to design the skeleton of your query, and then copy and modify the resulting SQL to suit your needs.

It is also possible to create a view from the Query Analyzer using SQL scripts. This technique is usually used when deploying a new database. The database structures, including views, are generated into SQL script files from the development database. The script is then imported using SQL Analyzer on the new server. When the script is run against an empty database, the database structure is then recreated in the new database.

The New View Window

The *New View*, or the *Design View*, window is the graphical tool provided in Enterprise Manager for the creation and modification of views. You can modify an existing view by right-clicking on it in Enterprise Manager. From the pop-up menu, choose Design View to bring up the Design View window. As shown in Figure C-2, the Design View window contains a toolbar.

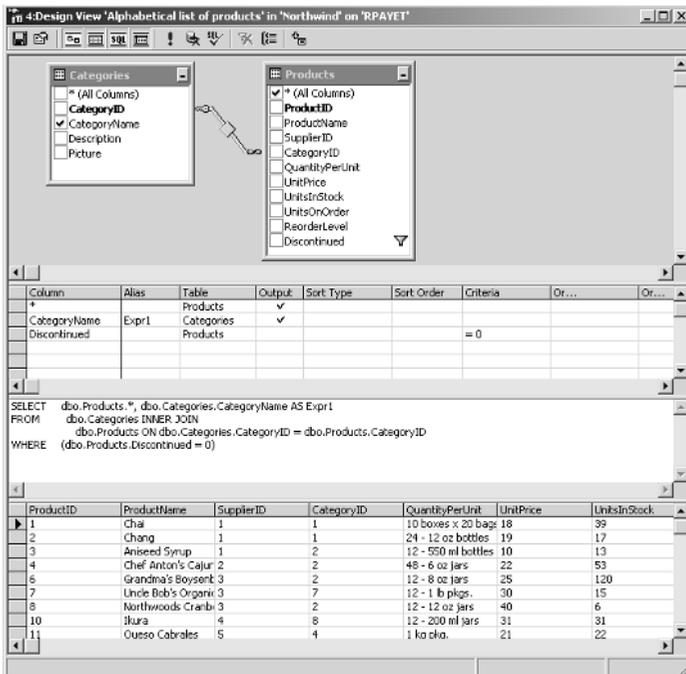


Figure C-2: The Design View window showing the "Alphabetical list of products" view



Note: The New View and Design View window are the same, except one is used for creating new views and the other is used for modifying an existing view.

The Toolbar

The toolbar has the following buttons (refer to Figure C-2) from left to right:

- **Save:** Saves the view. If the view is new, a dialog will pop up, allowing you to enter the name of the view.
- **Properties:** Opens a dialog box that allows you to set additional properties and options for the view. These include comments, encryption, schema binding, and with check option.
- **Show/Hide diagram pane:** Toggles the diagram pane on and off
- **Show/Hide Grid pane:** Toggles the grid pane on and off
- **Show/Hide SQL pane:** Toggles the SQL pane on and off
- **Show/Hide Results pane:** Toggles the results pane on and off
- **Run:** Runs the query in the SQL pane and displays results in the results pane
- **Cancel Execution and Clear results:** Can cancel the query that is running (if any) and clear the results pane. It is only active when there is a query running or the results pane is not empty.
- **Verify SQL:** Verifies that the query in the SQL pane is a valid query
- **Remove Filter:** Removes filter on specified columns. This is only active when a column with a filter is selected in the diagram pane. In Figure C-2, the Discontinued column in the Products table has a filter.

- **Use ‘Group by’:** Sets the Group by option so that you can use GROUP BY in the query. An additional column named Group by is added to the grid pane.
- **Add Table:** Adds a table to the diagram pane

Diagram Pane

The diagram pane shows a graphical layout of the view. This includes tables and links between related tables. Selected columns are also shown with a check mark next to the column heading. Right-clicking the pane background brings up the pop-up menu from which you can add new tables to the view. Right-clicking the tables and links brings up context-sensitive menus where you can set different options.

Grid Pane

The grid pane shows the selected columns in a grid format. From the grid pane, you can set additional options, such as Alias (name of the column in the view), Sort Type (Ascending or Descending), and Criteria (filter criteria). The section is converted into the WHERE clause of the SELECT statement).

SQL Pane

You can type the SQL query directly into the SQL pane. Any new tables referenced or any joins made will be reflected in the diagram and grid panes.

Results Pane

The results pane is used to display the result of running the SQL query. You can run the query by using the Run button on the toolbar.

Creating Views Using SQL

Although it is easier to design views using the Design View tool in Enterprise Manager, it is at times necessary to use T-SQL to express complex views or special view attributes.

Syntax

The syntax for creating, changing, and deleting a view is shown below.

Creating a view:

```
CREATE VIEW [ < database_name > . ] [ < owner > . ]
           < view_name >
           [ ( < column > [ ,...n ] ) ]
           [ WITH < view_attribute > [ ,...n ] ]
AS
< select_statement >
[ WITH CHECK OPTION ]
```

Changing a view:

```
ALTER VIEW [ < database_name > . ] [ < owner > . ]
           < view_name >
           [ ( < column > [ ,...n ] ) ]
           [ WITH < view_attribute > [ ,...n ] ]
AS
< select_statement >
[ WITH CHECK OPTION ]
```

Deleting a view:

```
DROP VIEW < view_name > [ ,...n ]
```



Note: Arguments between [] are optional.

Arguments

- < view_name >: The name of the view. View_name must follow the same convention as identifiers. The name can also include spaces, but in such a case, it must be enclosed between [].

- `< column >`: The name to be used for a column in the view. Specifying column name in CREATE VIEW is only required when a column is derived from a computed field or when there are columns in the SELECT part that have the same name (usually due to table joins that have columns with the same name). Column names can also be assigned in the SELECT statement. Specifying the view owner is optional.
- `,...n`: `n` is a placeholder indicating that multiple options can be specified.
- `< view_attribute >`: View attributes can include ENCRYPTION, SCHEMABINDING, and VIEW_METADATA.
 - ENCRYPTION: SQL Server stores the text for creating views in system table columns. This attribute option instructs SQL Server to encrypt the system table columns containing the text of the CREATE VIEW statement. This prevents the view from being published as part of the SQL Server replication. It also prevents the database user from being able to see the text for the CREATE VIEW statement.
 - SCHEMABINDING: This attribute binds the view to the schema. When used, select_statement must include the owner for any object referenced (owner.table_name, owner.view_name). Tables and views referenced in select_statement cannot be dropped (deleted) unless the bound view is dropped first. However, the referenced tables can be altered as long as the change does not affect the view definition.
 - VIEW_METADATA: Instructs SQL Server to return metadata information about the view, instead of metadata for base tables, to DBLIB, ODBC, and OLEDB APIs. This happens when

clients using DBLIB, ODBC, or OLEDB APIs request browse-mode metadata. Browse-mode metadata are additional metadata returned by SQL Server to client-side DBLIB, ODBC, or OLEDB APIs, allowing client-side APIs to implement updateable client-side cursors. With the VIEW_METADATA option, the browse-mode metadata returns the view name, as opposed to the base table names, when describing columns from the view in the result set.

- **Note:** When the VIEW_METADATA option is set, all columns (except for timestamp) of the view are updateable, as long as the view has INSERT or UPDATE INSTEAD OF triggers. Updateable views and triggers are discussed later in this chapter.
- <select_statement>: This is the SELECT statement or SQL query that defines the result set of the view. The SELECT statement can include references to tables, as well as other views. This can be any SQL SELECT statement, including multiple SELECT statements separated by the UNION operator; however, there are certain restrictions. The SELECT statement of a CREATE VIEW cannot include the following:
 - COMPUTE or COMPUTE BY clauses
 - ORDER BY clause, unless there is also a TOP clause in the select list
 - The into keyword
 - References to temporary tables or temporary table variables

✘ **Warning:** SQL Server allows a query expression to reference a maximum of 1,024 columns.

- WITH CHECK OPTION: Ensures that all data modification executed against the view adhere to the

criteria (usually set in the WHERE clause) set within select_statement. This ensures that the data modified remains visible through the view after the modification is committed.

✘ Warning: CREATE VIEW will fail if a view with the same name exists in the database. Existing views can be modified using ALTER VIEW instead of CREATE VIEW. The syntax is the same for both. It is also possible to drop an existing view and recreate it using CREATE VIEW.

Below is a sample script for creating the “Alphabetical list of products” view included with the Northwind database. Notice that the script first checks to see if the view already exists. If it exists, the view is then deleted using DROP VIEW.

Listing C-1: The “Alphabetical list of products” view

```

/*Drop the view if it exists*/
IF EXISTS (SELECT * FROM sysobjects
           WHERE id = object_id(N'[Alphabetical list of
           products]'))
           AND OBJECTPROPERTY(id, N'IsView') = 1)
           DROP VIEW [Alphabetical list of products]
GO
/*Create the view.*/
CREATE VIEW [Alphabetical list of products] AS
SELECT Products.*, Categories.CategoryName
FROM Categories INNER JOIN Products ON
           Categories.CategoryID = Products.CategoryID
WHERE Products.Discontinued = 0

```

▶ Note: Since the name of the view contains spaces, it must be enclosed in [].

Views in Practice

Now that you know the syntax and tools for creating views, let’s look at some examples of how to create views of your own. These scripts were designed for use with the Query Analyzer tool. You must connect to the

Northwind sample database before you execute the scripts. You can also use the New View window if you like. Just copy the query (the SELECT part only) into the SQL pane. The other pane will be automatically adjusted.

Example 1: Order Header Query

In this example, we are going to create a view from the Order table. As it is, the Order table is normalized. Although this is an efficient way to store data, it is not very intuitive for a user. So, we are going to present the user with a denormalized version of the table. Since the Order table represents the header of an order, we are going to call our view “Order Header Query.” The script for creating the view is shown below:

Listing C-2: Order Header Query view

```

/* Check if view already exists; if so, drop the view */
if exists
    ( select * from sysobjects
      where id = object_id(N'[Order Header Query]')
        and OBJECTPROPERTY(id, N'IsView') = 1 )
drop view [Order Header Query]
GO

CREATE VIEW [Order Header Query]
/*****
* Denormalized Order table to show values of linked *
* tables.                                           *
*****/
AS
SELECT Orders.OrderID as [Order ID],
       Customers.CompanyName as [Customer Company Name],
       Employees.Title + ': ' +
       Employees.FirstName + ' ' +
       Employees.LastName as Employee,
       Orders.OrderDate as [Order Date],
       Orders.RequiredDate [Required Date],
       Orders.ShippedDate as [Shipped Date],
       Shippers.CompanyName AS [Shippers Company Name],
       Orders.Freight,
       Orders.ShipName as [Ship Name],

```

```
Orders.ShipAddress as [Ship Address],
Orders.ShipCity as [Ship City],
Orders.ShipRegion as [Ship Region],
Orders.ShipPostalCode [Ship PostalCode],
Orders.ShipCountry as [Ship Country]
FROM Orders INNER JOIN
Customers ON Orders.CustomerID =
Customers.CustomerID INNER JOIN
Employees ON Orders.EmployeeID =
Employees.EmployeeID INNER JOIN
Shippers ON Orders.ShipVia = Shippers.ShipperID
```

First of all, the script checks and deletes a view with the same name if one already exists. Notice that the comment for the view is placed after the CREATE VIEW statement. This will ensure that the comment is kept as part of the view. Comments are important, especially for maintenance purposes.

In the query, column names are aliased to more user-friendly names, which include spaces. To use column names with spaces, the names must be enclosed in []. For example, OrderID is aliased as [Order ID].

```
SELECT Orders.OrderID as [Order ID],
```

CustomerID is replaced with a column named Customers.CompanyName from the linked Customers table and is aliased as [Customer Company Name]. ShipVia is also replaced by Shippers.CompanyName aliased as [Shippers Company Name]. The linked tables are shown in Figure C-3.

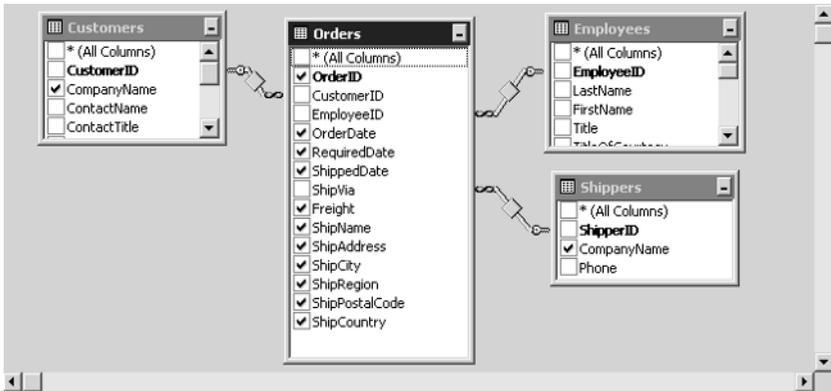


Figure C-3: Diagram pane for Order Header Query view

EmployeeID is replaced by a computed column, which combines three columns—Title, FirstName, and LastName—from the Employees table. Notice that colons followed by a space separate Title and FirstName. FirstName and LastName are separated by a space only.

```
Employees.Title + ': ' +
Employees.FirstName + ' ' +
Employees.LastName as Employee,
```

As you can see, when denormalizing (usually required for reporting purposes), you can combine multiple columns for the linked table into one view column.



Tip: To make your code more readable, remember to indent and put each column on a separate line. If a computed column is long, use multiple lines, but indent the following line farther.

Example 2: ABC Sales by Customer

Let's now look at a more complicated example. This example will show you how to use views as a stepping-stone for queries. Suppose you were given the task of designing an ABC Sales by Customer query. An ABC report is usually a performance analysis report that shows the top ten and bottom ten of whatever column

you are using for the performance analysis. In our case, we would use the sum of subtotals of orders for each customer. The query will show us our best and worst customers (in terms of sales). Usually in such cases, you would create two SELECT statements and then UNION the two, but because such a query requires the use of ORDER BY, UNION cannot be used. It must be done using two separate SELECT statements, as shown in Listing C-3:

Listing C-3: ABC Sales by Customer query

```

/*ABC Sales by Customer query*/
/*Top 10 Customers*/
SELECT TOP 10
    Orders.CustomerID,
    Customers.CompanyName,
    SUM(
        CONVERT ( money,
            (
                [Order Details].UnitPrice *
                [Order Details].Quantity
            ) *
            (
                1 - [Order Details].Discount
            ) / 100
        )
    ) AS Subtotal
FROM    [Order Details] INNER JOIN
        Orders ON [Order Details].OrderID =
            Orders.OrderID INNER JOIN
        Customers ON Orders.CustomerID =
            Customers.CustomerID INNER JOIN
        Employees ON Orders.EmployeeID =
            Employees.EmployeeID
GROUP BY Orders.CustomerID, Customers.
        CompanyName, [Order Details].OrderID
ORDER BY Subtotal DESC

/*Bottom 10 Customers*/
SELECT TOP 10
    Orders.CustomerID,
    Customers.CompanyName,

```

```

SUM(
    CONVERT ( money,
        (
            [Order Details].UnitPrice *
            [Order Details].Quantity
        ) *
        (
            1 - [Order Details].Discount
        ) / 100
    )
    * 100
) AS Subtotal
FROM [Order Details] INNER JOIN
Orders ON [Order Details].OrderID =
Orders.OrderID INNER JOIN
Customers ON Orders.CustomerID =
Customers.CustomerID INNER JOIN
Employees ON Orders.EmployeeID =
Employees.EmployeeID
GROUP BY Orders.CustomerID, Customers.
CompanyName, [Order Details].OrderID
ORDER BY Subtotal ASC

```

Notice that the only difference between the two SELECT statements is the sorting order.

```
ORDER BY Subtotal ASC
```

and

```
ORDER BY Subtotal DESC
```

This is not ideal because we get two separate result sets. Let's see if we can find a solution to the problem using views. The Northwind database already has a view called Order Subtotals. We shall use it to do our ABC views. You can double-click on the view to see the scripts that were used to create it, or you can use the Design View window.

 **Note:** Double-click on the Order Subtotals view to bring up the View Properties window. Right-click on the view and choose Edit View from the pop-up to bring up the Design View window.



Warning: Do not alter the views that came with the Northwind database. This might cause some of the scripts included in this book to fail.

All we need to do now is create two additional views: Top 10 Customers and Bottom 10 Customers. Both will use the Order Subtotals window. Listing C-4 shows the required scripts.

Listing C-4: Top 10 Customers and Bottom 10 Customers view

```

/* Check if view already exists; if so, drop the view */
if exists
  ( select * from sysobjects
    where id = object_id(N'[Top 10 Customers]')
    and OBJECTPROPERTY(id, N'IsView') = 1 )
drop view [Top 10 Customers]
GO

CREATE VIEW [Top 10 Customers]
/*****
* Show top best Customers in      *
* term of sales                   *
*****/
AS
SELECT TOP 10
    Orders.CustomerID,
    Customers.CompanyName,
    SUM([Order Subtotals].Subtotal) AS Subtotal
FROM Customers INNER JOIN
    Orders ON Customers.CustomerID = Orders.CustomerID
    INNER JOIN
    [Order Subtotals] ON Orders.OrderID =
    [Order Subtotals].OrderID
GROUP BY Orders.CustomerID, Customers.CompanyName
ORDER BY [Order Subtotals].Subtotal DESC
GO

/* Check if view already exists; if so, drop the view */
if exists
  ( select * from sysobjects
    where id = object_id(N'[Bottom 10 Customers]')
    and OBJECTPROPERTY(id, N'IsView') = 1 )
drop view [Bottom 10 Customers]

```

```

GO

CREATE VIEW [Bottom 10 Customers]
/*****
* Show top worst Customers in      *
* term of sales                    *
*****/
AS
SELECT TOP 10
    Orders.CustomerID,
    Customers.CompanyName,
    SUM([Order Subtotals].Subtotal) AS Subtotal
FROM Customers INNER JOIN
    Orders ON Customers.CustomerID = Orders.CustomerID
    INNER JOIN
    [Order Subtotals] ON Orders.OrderID = [Order
    Subtotals].OrderID
GROUP BY Orders.CustomerID, Customers.CompanyName
ORDER BY [Order Subtotals].Subtotal ASC

```

Once these views have been created, we can then use them to create the new ABC Sales by Customer query.

Listing C-5: The new ABC Sales by Customer query

```

/*ABC Sales by Customer*/
SELECT * from [TOP 10 Customers]
UNION
SELECT * from [Bottom 10 Customers]

```

As you can see, this query is much simpler than the one in Listing C-3. The problem has been broken down into simpler steps and is also the complete solution. In this case, the full solution is not possible without the views.

Updateable Views

Views can be used as tables, and the underlying tables of the view can be updated through the view provided that:

- The view contains at least one table in the FROM clause of the view definition. You cannot update a view that is based only on expressions.

- No aggregate functions (AVG, COUNT, SUM, MIN, MAX, GROUPING, STDEV, STDEVP, VAR, VARP) or GROUP BY, UNION, DISTINCT, or TOP clauses are used in the select list. The only instance when an aggregate function allows an update is if the function is used within a subquery defined in the FROM clause and provided that the derived values from the function are not modified.
- No derived columns are used in the select list.

Partitioned views using the UNION ALL operator can be updateable. Any views that do not meet these criteria can still be made updateable through the use of INSTEAD OF triggers. Triggers are explained in a later section.

Stored Procedures

Now that you know how to create views, let's move on to stored procedures. So the first question beckons: What is a stored procedure?

Introduction to Stored Procedures

A *stored procedure* is a precompiled collection of Transact-SQL statements stored with the database and processed as a logical unit. SQL Server comes pre-loaded with its own sets of stored procedures used for managing the SQL Server and displaying information about databases and users. These supplied stored procedures are called *system stored procedures*. System stored procedures are used for database administrative tasks and are stored in the system database called Master. For example, you can use the sp_help system stored procedure to get information about objects in the current database.

 **Note:** The maximum size of a stored procedure is 128MB.

Advantages of Stored Procedures

There are many advantages in putting T-SQL code in stored procedures rather than embedding it in a client application.

Modular Programming

The stored procedure can be written once and stored with the database. This is often done by database experts and is a way of optimizing database operations. A stored procedure can be modified independently of the client application source code. Multiple client applications written in a variety of programming languages (e.g., VB .NET, C#, C++, ASP .NET, and Java) can use the same stored procedures. This helps avoid duplication of effort. It also allows customization of some aspects of the application without the need for change in the client's source code.

Optimized Execution

Before any query can be executed by the server, the T-SQL codes it contains are parsed and an execution plan is created. The execution plan is an understanding of what the query wants to do, and it effectively guides the SQL Server to the most efficient way to execute the query.

When a stored procedure is created, an execution plan for the T-SQL code it contains is created, optimized, and stored with the procedure. Subsequently, when the stored procedure is executed, the same query plan is used. The stored procedure is also cached in memory after it is executed for the first time.

On the other hand, when a client application sends a query to the database, SQL Server parses the query to create an execution plan. The query is not cached, and the execution plan is recalculated each time the client sends the query. This makes stored procedures much more efficient, especially for long sets of T-SQL codes and repetitive codes such as those found in a loop.

Reduced Network Traffic

An operation that requires hundred of lines of T-SQL statements can be achieved by executing just one line of code that calls a stored procedure. This reduces network traffic considerably, both in terms of what is sent to the server and also what is sent back to the client application. It also has the added bonus of reducing the amount of code in the client application, thus making it easier to maintain and follow.

Enhanced Security

A user can be given access to execute a given stored procedure, even if he does not have permission to execute a T-SQL statement found in the procedure. This way, you can allow users to update the data only through stored procedures where you can validate and implement business rules.

Managing Stored Procedures

Stored procedures are created either within the Enterprise Manager using the Stored Procedure Properties window or in SQL Query Analyzer.

Click on the Stored Procedures node of the Northwind database in the console tree. All the stored procedures for Northwind are loaded inside the right-hand pane of the Enterprise Manager. Right-click the Stored Procedures node and click New Stored Procedure in the pop-up menu. The Stored Procedure Properties window

opens. The Text field is where the logic for the procedure is coded. You can also edit existing stored procedures by double-clicking on the stored procedure name in the right pane. You cannot edit system stored procedures, but you can still see their code.

► **Note:** Once you have opened the Stored Procedure Properties window, you must close it before you can use any other part of Enterprise Manager.

You will find that working in SQL Query Analyzer is much easier. The Query Analyzer contains some useful tools that will help you with any T-SQL statements. For instance, you can see the execution plan that will warn of any bottleneck in the code.

► **Note:** In SQL 2000, the Query Analyzer contains an object browser.

You can use the object browser to help code your stored procedures. You simply drag the object, which also includes T-SQL functions, to your programming window. The object browser can be toggled on and off from the menu by selecting Tools, Object Browser, Show/Hide. Alternatively, you can use F8 as a keyboard shortcut.

Creating Stored Procedures

The code for stored procedures is written in T-SQL. To effectively write complicated stored procedures, you must possess a firm knowledge of T-SQL.

Syntax

Create a stored procedure:

```
CREATE PROC[EDURE] < procedure_name > [ ; number ]
    [ { @parameter data_type }
      [ VARYING ] [ = default ] [ OUTPUT ]
    ] [ ,...n ]
    [ WITH < procedure_attribute > [ ,...n ] ]
    [ FOR REPLICATION ]
```

```
AS
< sql_statement > [ ...n ]
```

Modify a stored procedure:

```
ALTER PROC[EDURE] < procedure_name > [ ; number ]
  [ { @parameter data_type }
    [ VARYING ] [ = default ] [ OUTPUT ]
  ] [ ,...n ]
  [ WITH < procedure_attribute > [ ,...n ] ]
  [ FOR REPLICATION ]
AS
< sql_statement > [ ...n ]
```

Delete a stored procedure:

```
DROP PROCEDURE < procedure_name > [ ,...n ]
```



Note: Arguments between [] are optional. Those between { } must appear together.

Arguments

- < procedure_name >: The name of the stored procedure. The name can also include spaces, but if so, it must be enclosed in []. Stored procedure names must follow the same conventions as identifiers.
- ; number: This is an optional number used to group procedures of the same name so that they can be deleted at the same time with a single DROP PROCEDURE statement. For example, you can create a set of procedures for samples in this book as sampleproc;1, sampleproc;2, and so on. After you have finished with the samples, you can delete the lot with one statement: DROP PROCEDURE sampleproc.
- @parameter: This is the name of a parameter to the procedure. Parameter names must start with a @ and must follow the same conventions as identifiers. One or more parameters can be declared with a stored procedure. When the procedure is called, the

value for each must be supplied, unless a default is defined. Stored procedures in SQL Server 2000 can have a maximum of 2,100 parameters.

Parameters are local to the procedure in which they are defined.

- **data_type:** This is the parameter data type. You can use any MS SQL 2000 data types, including text, ntext, and image. The cursor data type can be used only on OUTPUT parameters. When you specify cursor parameters, the VARYING and OUTPUT keywords must also be specified.

 **Note:** Cursor data types in parameters contain a reference to a cursor. In SQL, a cursor is a special object that works on result sets. Its functionality is similar to ADO .NET, and it allows row-by-row manipulation of the result set. Cursors are created using the CREATE CURSOR statement.

- **VARYING:** This applies only to cursor parameters and specifies that the result set is supported as an OUTPUT parameter. It denotes that the output is constructed dynamically by the stored procedure and its contents can vary.
- **Default:** This is the default value for the parameter. The procedure can be executed without specifying a value (in which case the default value is assigned to the parameter). The default must be a constant or it can be null. It can also include strings with wildcard characters (% , _ , [], and [^]) if the procedure uses the parameter with the Like keyword.
- **OUTPUT:** This indicates that the parameter will return a value. The value can be returned to EXEC[UTE]. OUTPUT parameters are used to return information to the calling procedure. Text, ntext, and image parameters can be used as

OUTPUT parameters. If the data type is a cursor, the OUTPUT option must be used.

- `,...n:` `n` is a placeholder indicating more than one option can be specified.
- `< procedure_attribute >`: Procedure attributes are RECOMPILE and ENCRYPTION.
 - RECOMPILE: Directs SQL Server to not cache a plan for this procedure, and the procedure is recompiled at run time. Use the RECOMPILE option when using temporary values without overriding the execution plan cached in memory.
 - ENCRYPTION: SQL Server stores the text for creating views in system table columns. This attribute option instructs SQL Server to encrypt the system table columns containing the text of the CREATE PROC[EDURE] statement. This prevents the stored procedure from being published as part of SQL Server replication. It also prevents database users from being able to see the code for the stored procedure.
- FOR REPLICATION: Creates a stored procedure that is used as a stored procedure filter and is only executed during replication. This option cannot be used with the RECOMPILE option.
- `<sql_statement>`: This is the T-SQL code that forms part of the procedure. This can also include calls to other stored procedures. The compiler will include all the SQL code in the script as part of the stored procedure. You can, however, indicate where the stored procedure ends by using the GO statement.



Note: The syntax for ALTER PROC[EDURE] is the same as for CREATE PROC[EDURE].

Executing Stored Procedures

Stored procedures are usually called from the client; however, it is possible to issue a call to a stored procedure from within another stored procedure or trigger.

The syntax for executing a stored procedure is:

```
EXECUTE sp_name [@parameter = value] [, ...n]
```

`sp_name` is the name of the stored procedure you want to execute. If the stored procedure takes any parameter, you should also pass a valid value to it. It is possible to pass the parameter values separated by commas. This is practical only when you know the order in which the parameters occur in the stored procedure. It is a better programming practice to include the `@parameter = value` syntax. This makes the code more readable.

The tool you will use most to create your stored procedure is the Query Analyzer. The new version has some templates included, which you can use as a building block for your stored procedure.

Stored Procedures in Action

Now that you know how to create a stored procedure, it is time we look at an example.

 **Note:** By convention, most stored procedure names are prefixed with `sp` and do not have spaces. However, you can use your own convention including spaces.

Example: Subtotal of an Order

In this example, our aim is to define a stored procedure that returns the customer ID and subtotal of an order. We will call the stored procedure `sp_SubTotal`.

► **Note:** Stored procedures can only return an integer directly. Use this primarily to denote error codes. To return other types, use the OUTPUT parameter. Result set can also be returned if any SQL statement does so.

Listing C-6: sp_SubTotal

```
-- =====
-- Create procedure sp_SubTotal
-- =====
-- Drop the procedure if it already exists

IF EXISTS (SELECT name
           FROM   sysobjects
           WHERE  name = N'sp_SubTotal'
           AND   type = 'P')
  DROP PROCEDURE sp_SubTotal
GO

-- creating the stored procedure version one
-- The values are returned as result sets
CREATE PROCEDURE sp_SubTotal;1
  @OrderID integer
AS

  SELECT Orders.CustomerID,
         [Order Subtotals].Subtotal
  FROM   Orders INNER JOIN
         [Order Subtotals] ON Orders.OrderID = [Order
         Subtotals].OrderID
  WHERE  Orders.OrderID = @OrderID

GO

-- creating the stored procedure second version
-- The values are returned through OUPUT parameters
CREATE PROCEDURE sp_SubTotal;2
  @OrderID integer,
  @CustomerID nchar(5) OUTPUT,
  @SubTotal money OUTPUT
AS
```

```

SELECT @CustomerID = Orders.CustomerID,
       @SubTotal = [Order Subtotals].Subtotal
FROM   Orders INNER JOIN
       [Order Subtotals] ON Orders.OrderID = [Order
       Subtotals].OrderID
WHERE  Orders.OrderID = @OrderID

GO

-- =====
-- example to execute the stored procedure
-- =====

DECLARE @ID int
DECLARE @CustomerID nvarchar(5)
DECLARE @SubTotal money

SET @ID = 10249

-- Execute version 1, The default
EXECUTE sp_SubTotal @ID

-- Execute the second version
EXECUTE sp_SubTotal;2 @ID, @CustomerID OUTPUT, @SubTotal
OUTPUT
PRINT @CustomerID
PRINT CAST(@SubTotal as varchar)
GO

```

The script above produces two stored procedures that perform the same calculation. The difference is that version one returns a result set (same as if the query was run directly), and the second version returns values through OUTPUT parameters.

Triggers

There are times when, for data integrity purposes, you need to maintain certain rules. These could be business rules, such as conditions that need to be met before allowing updates of certain fields. This is when triggers come in handy.

Introduction to Triggers

A *trigger* is a special kind of stored procedure that executes automatically when a change occurs in a database record. There are three specific types of events that triggers are mainly concerned with: INSERT, UPDATE, and DELETE. Triggers allow you to run code automatically that can perform any required check, change other required data, or even reject the change if certain conditions are not met.

Being specific to the type of operation being performed on a record, UPDATE, INSERT, and DELETE are powerful tools when it comes to the maintenance of referential integrity and enforcement of business rules. Triggers can be defined on tables or views. In SQL Server 2000, there is also a new kind of trigger called INSTEAD OF. These triggers are executed instead of the triggering action. For example, instead of doing an INSERT, UPDATE, or DELETE operation on the table, the control is passed to the INSTEAD OF trigger, which is then responsible for performing the appropriate action. INSTEAD OF triggers can be defined on views referencing multiple tables (these views are normally not updateable), so each of the referenced tables can be updated appropriately.

Advantages of Triggers

A trigger is essentially a stored procedure, so it can contain all the logic any stored procedure can handle. It is a good programming practice to code logic inside stored procedures. Then, inside the triggers, you use the stored procedure as needed. Since triggers are a special kind of stored procedure, they have the same advantages in terms of performance and speed as stored procedures. They also have some additional benefits, including enforcing business rules and automatically running routines.

Enforce Business Rules on the Database

Triggers are used to enforce business rules. The rules are enforced on the database and independent of client applications. This means that the rules are always enforced, no matter what the client application does. You can even have different client applications, such as a web application client and a Windows Form application client. If the rules change in the future, they can be modified without having to change the client application.

Automatic Invocation of Other Routines

Triggers can also be used to automatically run certain routines when there is a change to a table. These are usually for auditing purposes, and the routine stores any important changes in an audit trail table. For example, it might be required that an audit trail is kept whenever an employee salary is changed. Triggers can also be used to automatically send e-mails to users whenever certain changes occur.

Managing Triggers

As with stored procedures, the main tool for creating triggers is the SQL Analyzer. However, you can also view existing triggers from the Enterprise Manager. To do this, simply right-click on a table and select All Tasks, Manage Trigger from the pop-up menu. This will bring up the Trigger Properties window.

Creating Triggers

Just like stored procedures, you also need to know T-SQL before you can create a trigger. There are some T-SQL statements that are only applicable to triggers. Let's have a look at the T-SQL syntax for managing triggers.

Syntax

The full SQL syntax for creating, changing, and deleting triggers is shown below.

Create a trigger:

```
CREATE TRIGGER < trigger_name >
ON { table | view }
[ WITH ENCRYPTION ]
{
  { { FOR | AFTER | INSTEAD OF } { [DELETE] [,] [INSERT]
    [,] [UPDATE] }
    [ WITH APPEND ]
    [ NOT FOR REPLICATION ]
  }
AS
  < trigger_logic > [ ...n ]
```

Modify a trigger:

```
ALTER TRIGGER < trigger_name >
ON { table | view }
[ WITH ENCRYPTION ]
{
  { { FOR | AFTER | INSTEAD OF } { [DELETE] [,] [INSERT]
    [,] [UPDATE] }
    [ WITH APPEND ]
    [ NOT FOR REPLICATION ]
  }
AS
  < trigger_logic > [ ...n ]
```

Delete a trigger:

```
DROP TRIGGER < trigger_name >
```

Arguments

- **< trigger_name >**: The name of the trigger. The name can also include spaces, but if so, it must be enclosed in []. Trigger names must follow the same conventions as identifiers.

- **WITH ENCRYPTION:** SQL Server stores the text for creating the trigger in system tables. This attribute option instructs the SQL Server to encrypt the system table columns containing the text for the trigger. This prevents the trigger from being published as part of the SQL Server replication. It also prevents the database user from being able to see the code for the trigger.
- **AFTER:** This is new in SQL Server 2000. It is used to specify that the trigger is fired only when all operations specified in the triggering SQL statement have executed successfully. This trigger will only execute after all referential cascade actions and constraint checks succeed.



Note: Using the FOR keyword has the same effect as using the AFTER keyword. AFTER triggers cannot be defined on views.

- **INSTEAD OF:** This specifies that the trigger is executed instead of the triggering SQL statement and, therefore, overrides the actions of the triggering statements. At most, there can only be one INSTEAD OF trigger per INSERT, UPDATE, or DELETE statement defined on a table or view. However, it is possible to define views on views where each view has its own INSTEAD OF trigger.



Warning: INSTEAD OF triggers are not allowed on updateable views with the check option set.

- { [DELETE] [,] [INSERT] [,] [UPDATE] }: These are the keywords that specify which data modification statements, when attempted against this table or view, will activate the trigger. At least one option must be specified. You can have any combination of these in the trigger definition.

- **Note:** For **INSTEAD OF** triggers, the **DELETE** option is not allowed on tables that have a referential relationship specifying a cascade action **ON DELETE**. Also, the **UPDATE** option is not allowed on tables that have a referential relationship specifying a cascade action **ON UPDATE**.
- **WITH APPEND:** This is used to add additional triggers of an existing type. **WITH APPEND** is used for older versions of SQL Server and the clause is not needed to add an additional trigger in SQL 2000. In future versions it will not be supported; thus, you should not use it.
 - **NOT FOR REPLICATION:** Prevents the trigger from being executed when a replication process modifies the table
 - **< trigger_logic >:** This includes the SQL statements and logic for the trigger. There are logical tables and special logical statements that you can use in triggers. These are deleted and inserted logical tables, and the logical clause **IF UPDATE (column)**.
 - **deleted and inserted:** These are logical (conceptual) tables. They are structurally similar to the table on which the trigger is defined and hold the old values or new values of the rows that may be changed by the action. For example, to retrieve all values in the deleted table, use **SELECT * FROM deleted**.
 - **IF UPDATE (column):** Tests for an **INSERT** or **UPDATE** action to a specified column and is not used with **DELETE** operations. More than one column can be specified. Because the table name is specified in the **ON** clause, there is no need to include the table name before the column name in an **IF UPDATE** clause. **IF UPDATE** will return the **TRUE** value in **INSERT** actions because the

columns have either explicit values or implicit (null or default) values inserted.

Recursive Triggers

SQL Server also allows recursive invocation of triggers when the recursive triggers setting is enabled in the database.

Recursive triggers allow two types of recursion to occur:

- Indirect recursion
- Direct recursion

With indirect recursion, an application updates table one, which fires trigger one, updating table two. Trigger two then fires and updates table one, and so on.

With direct recursion, the application updates a table, which fires a trigger, updating the table itself. Because the table was updated again, the trigger fires again, and so on.

 **Note:** The above behavior occurs only if the recursive triggers setting of `sp_dboption` is enabled. There is no defined order in which multiple triggers specified for a given event are executed. Each trigger should be self-contained.

Disabling the recursive triggers setting only prevents direct recursions. To disable indirect recursion as well, set the nested triggers server option to 0 using `sp_configure`.

If any of the triggers do a `ROLLBACK TRANSACTION`, regardless of the nesting level, no further triggers are executed.

Nested Triggers

Triggers can be nested to a maximum of 32 levels. If a trigger changes a table on which there is another trigger, the second trigger is activated and can then call a third trigger, and so on.



Warning: If any trigger in the chain sets off an infinite loop, exceeding the nesting level, the trigger is canceled. To disable nested triggers, set the nested triggers option of `sp_configure` to 0 (off). The default configuration allows nested triggers. If nested triggers is off, recursive triggers is also disabled, regardless of the recursive triggers setting of `sp_dboption`.

Triggers in Action

Now that you know how to create triggers, let's see an example.

Example: Prevent Ordering of Discontinued Products

When ordering new products, you want to make sure that no orders are placed for discontinued products. To do this, you must create an UPDATE/INSERT trigger on the Order Details table.

Listing C-7: Check Product Order Details trigger

```
-- =====
-- Drop trigger if it already exists
-- =====

IF EXISTS (SELECT name
           FROM   sysobjects
           WHERE  name = N'Check Product Order Details'
           AND   type = 'TR')
    DROP TRIGGER [Check Product Order Details]
GO

-- =====
```

```

-- Create trigger [Check Product Order Details]
-- =====

CREATE TRIGGER [Check Product Order Details]
ON [Order Details]
FOR INSERT, UPDATE
AS
BEGIN

-- =====
-- Do the Discontinued check only if ProductID
-- has been modified
-- =====

IF UPDATE(PRODUCTID)
BEGIN

    DECLARE @DISCONTINUED BIT
    DECLARE @PRODUCTID integer

-- =====
-- Set discontinued on as default, so that if it
-- does not exist, there will also be an error.
-- This is for illustration only. This test is
-- already done in the foreign key constraint
-- =====
    SET @DISCONTINUED = 1

-- =====
-- Get the Discontinued and ProductID
-- Note the use of logical table "inserted" to
-- work out the ProductID
-- =====
    SELECT @DISCONTINUED = Discontinued,
           @PRODUCTID = P.ProductID
    FROM Products P INNER JOIN Inserted I
    ON P.ProductID = I.ProductID

-- =====
-- IF product is discontinued, then RAISE an
-- error and rollback transaction to prevent saving
-- =====
    IF @DISCONTINUED = 1
    BEGIN
        RAISERROR

```

```
        (' The Product(ID: %d) is discontinued or does not  
        exist.', 16, 1, @PRODUCTID)  
    ROLLBACK TRANSACTION  
END  
END  
END  
GO
```

The above script example demonstrates the use of a simple trigger. You can test the trigger by changing the value of the ProductID in an existing row in the Order Details table to that of a discontinued product. Figure C-4 shows the resulting error message that is generated if the product is discontinued.

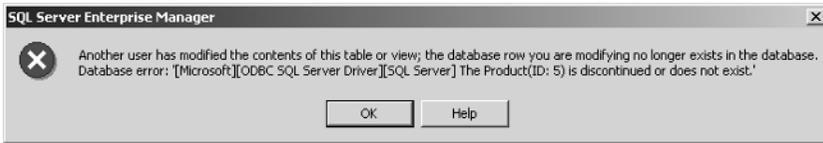


Figure C-4: Error generated by trigger

Summary

In this chapter you learned about the following database objects:

- Views
- Stored procedures
- Triggers

Although you are able to create all of these objects using SQL code, SQL Server provides a number of visual tools to help you achieve any task you may decide to undertake with these objects. The SQL Query Analyzer is a powerful T-SQL development tool that you can use to write and test your SQL scripts.

Views, stored procedures, and triggers can be used together to generate a powerful database solution that hinges on efficient business rule enforcement.

Advanced SQL Query Techniques

In This Appendix

This appendix will show you various tricks and techniques that you can use with SQL queries.

These include:

- Advanced insertion and deletion
- Recursive stored procedures
- Dynamic queries
- Stored procedure generator
- GROUP BY with CUBE and ROLLUP
- Server cross tabulations with SQL

 **Note:** In this appendix, all examples will use the Northwind database, as is the case throughout the book. Some of the examples in this appendix will cause deletion of data. It is, therefore, recommended that you back up the Northwind database so that you can restore it after you are done with this appendix.

Advanced Insertion and Deletion

Traditionally with SQL, INSERT and DELETE statements work with only one row at a time. For example, if you have ten rows to insert in a table, you have to do ten INSERT statements, usually in a loop. This can be very inefficient.

Insertion with SELECT

Standard insertion is very simple. For example, let's consider the case where you want to add a product to an order. To simplify error checking, we will create a stored procedure called sp_OrderAddProduct.

Listing D-1

```

/*****
Drop stored procedure if already exist
*****/
if exists (select * from dbo.sysobjects where id =
          object_id(N'sp_OrderAddProduct') and
          OBJECTPROPERTY(id, N'IsProcedure') = 1)
drop procedure sp_OrderAddProduct
GO

create procedure sp_OrderAddProduct
/*****
Procedure:   sp_OrderAddProduct
Date:       14 July 2002
Author:     Ryan N. Payet
Description: Insert New products in [Order Details] table

Version History
-----
0.1, 14 July 2002, Ryan N. Payet:
        Procedure Created

*****/
@OrderID integer,
@ProductID integer,
@Quantity smallint = 1,

```

```

@Discount real = 0
as
BEGIN

    DECLARE @@UnitPrice as Money
    DECLARE @@ERRORMSG as varchar(256)

    /*Check Order exist */
    IF NOT EXISTS (SELECT OrderID FROM Orders WHERE OrderID
        = @OrderID )
    BEGIN
        /*Order does not exist, raise error*/
        SET @@ERRORMSG = 'Given Order does not exist in table
            Orders'
        RAISERROR ( @@ERRORMSG, 16, 1)
        RETURN 1
    END

    /*Check Valid Product exist */
    IF NOT EXISTS
    (
        SELECT ProductID FROM Products
        WHERE ProductID = @ProductID AND
        Discontinued = 0
    )
    BEGIN
        /*Order does not exist, return with 1*/

        SET @@ERRORMSG = 'Given Product does not exist in table
            Products or it has been discontinued'

        RAISERROR ( @@ERRORMSG , 16, 1)
        RETURN 1
    END

    SELECT @@UnitPrice = UnitPrice
    FROM Products
    WHERE ProductID = @ProductID
    /* Do the insertion */
    BEGIN TRANSACTION

        INSERT INTO [Order Details]
            (OrderID, ProductID, UnitPrice, Quantity, Discount)

```

```

VALUES
    ( @OrderID, @ProductID, @@UnitPrice, @Quantity,
      @Discount)

IF @@ERROR <> 0
    ROLLBACK TRANSACTION
ELSE
    COMMIT TRANSACTION

RETURN 0

END

GO

```



Tip: Notice that there is a header comment within the CREATE statement that describes the functionality and version history of stored procedures. This allows the comments to be stored together with the procedure in the database so that future developers will be able to see the comment, even though they might not have the source script.

The stored procedure is simple and contains some error checking routines. This error checking helps to maintain data integrity, such as preventing addition of discontinued products, and avoid getting constraint violation errors with foreign keys. To do error checking and get the current unit price of a product requires three SELECT statements. Now consider the following listing, which makes use of INSERT with SELECT.

Listing D-2

```

/*****
Drop stored procedure if already exist
*****/
if exists (select * from dbo.sysobjects where id =
           object_id(N'sp_OrderAddProduct') and
           OBJECTPROPERTY(id, N'IsProcedure') = 1)
drop procedure sp_OrderAddProduct
GO

```

```

create procedure sp_OrderAddProduct
/*****
Procedure:    sp_OrderAddProduct
Date:        14 July 2002
Author:      Ryan N. Payet
Description:  Insert New products in [Order Details] table

Version History
-----
0.1, 14 July 2002, Ryan N. Payet:
    Procedure Created
0.2, 14 July 2002, Ryan N. Payet:
    Optimized to use INSERT with SELECT
*****/
@OrderID integer,
@ProductID integer,
@Quantity smallint = 1,
@Discount real = 0
as
BEGIN

    DECLARE @@ERRORMSG as varchar(256)

    /* Do the insertion */
    BEGIN TRANSACTION

        INSERT INTO [Order Details]
            (OrderID, ProductID, UnitPrice, Quantity, Discount)
        SELECT Orders.OrderID, Products.ProductID,
            Products.UnitPrice,1,0
        FROM Orders, Products
        WHERE Orders.OrderID = @OrderID AND
            Products.ProductID = @ProductID AND
            Products.Discontinued = 0

    /*
    Check to see if 1 row was affected
    If no row or more than 1 row was inserted
    then there was error
    */
    IF @@ROWCOUNT <> 1
    BEGIN

```

```

SET @@ERRORMSG = 'Given Order or Product does not
                  exist or the Product has been
                  discontinued'
RAISERROR ( @@ERRORMSG, 16, 1)
ROLLBACK TRANSACTION
RETURN 1
END
ELSE
BEGIN
  COMMIT TRANSACTION
  RETURN 0
END

END

GO

```

Version 0.2 of the stored procedure is smaller, and the INSERT statement now uses a SELECT to generate the data it requires for the insert. This is much more efficient than the previous version, which required three SELECT statements and one INSERT. The main rule to remember is that the columns returned by the SELECT statement must be compatible with the column list in the INSERT INTO section. Another variation of this is to use a stored procedure that returns Recordset.

```

INSERT INTO <Table_Name>
(<Column_List>)
EXECUTE <Stored_Procedure>

```

The INSERT with SELECT really excels when there is more than one row of data that you want to insert. Suppose you want to copy the order details of one order to another. With the standard INSERT, you will have to loop through the source and do an insert for each row. Using INSERT with SELECT, only one INSERT statement is required.

Listing D-3

```

DECLARE @@SrcOrderID as integer
DECLARE @@DestOrderID as integer

```

```
SET @@SrcOrderID = 10248
SET @@DestOrderID = 10296

INSERT INTO [Order Details]
    (OrderID, ProductID, UnitPrice, Quantity, Discount)
SELECT @@DestOrderID,
    [Order Details].ProductID,
    [Order Details].UnitPrice,
    [Order Details].Quantity,
    [Order Details].Discount
FROM [Order Details]
WHERE [Order Details].OrderID = @@SrcOrderID AND
    [Order Details].ProductID NOT IN
    ( SELECT ProductID FROM [Order Details]
      WHERE [Order Details].OrderID = @@DestOrderID
    )
```

In the example in Listing D-3, the order details of order 10248 are copied to order 10296. If you look carefully at the SELECT statement, you will notice that it includes a sub-SELECT in the WHERE clause to prevent copying of products that are already available in order 10296.

DELETE with Multiple Table WHERE Clause

We have so far seen how we can insert multiple-row data using SELECT. With DELETE, you have a WHERE clause, so it is possible to delete more than one row of data. You are probably wondering when you would need to refer to multiple tables. Consider the following problem. Suppose you want to delete discontinued products from the Order Details table that has been maintained by a particular employee. The relationship between the Orders, Order Details, and Products tables are shown in Figure D-1.

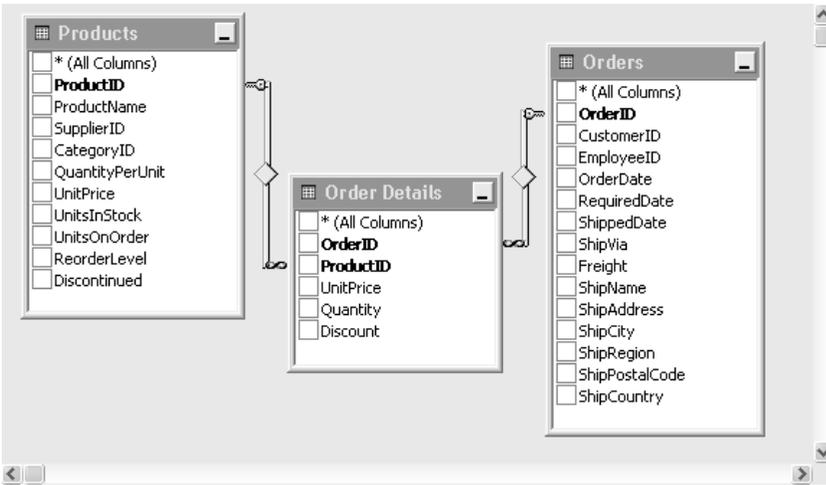


Figure D-1: Orders, Order Details, and Products tables

To solve the problem, let's approach it from a SELECT angle, which is "Show discontinued Products from Order Details that has been maintained by a particular employee." Using the relationship, we can do a simple SELECT, as shown here:

Listing D-4

```
SELECT [Order Details].OrderID,
       [Order Details].ProductID
FROM   [Order Details]
INNER JOIN Products
       ON [Order Details].ProductID = Products.ProductID
INNER JOIN Orders
       ON Orders.OrderID = [Order Details].OrderID
WHERE  (Products.Discontinued = 1) AND
       (Orders.EmployeeID = 1)
```

Note: For the example, we will use the employee with EmployeeID 1. This can be replaced later by the required EmployeeID. Discontinued products have the Products.Discontinued value of 1.

The result of the SELECT is the rows that we want to delete. By doing the SELECT first, we are able to see exactly which row will be deleted. Now it is a simple matter of modifying the script to make it do the deletion.

Listing D-5

```
DELETE [Order Details]
FROM   [Order Details]
       INNER JOIN Products
          ON [Order Details].ProductID = Products.ProductID
       INNER JOIN Orders
          ON Orders.OrderID = [Order Details].OrderID
WHERE  (Products.Discontinued = 1) AND
       (Orders.EmployeeID = 1)
```



Warning: Running the script will cause certain data to be deleted. It is recommended that you back up the Northwind database before you run the script. You can then restore it later when you have finished with this appendix.

One additional point about DELETE is that if the table is used in a sub-query, you must use an alias. For example, if you want to delete the five lowest performing Order Details, you would have to use the following script:

Listing D-6

```
DELETE [Order Details]
FROM   ( SELECT TOP 5 OrderID,
          SUM(UnitPrice * Quantity * (1 - Discount)) as
          Ordervalue
        FROM [Order Details] GROUP BY OrderID
        ORDER BY Ordervalue
        ) as T1
WHERE  [Order Details].OrderID = T1.OrderID
```

As you can see, you can even use TOP, GROUP BY, and ORDER BY in the sub-SELECT.

Recursive Stored Procedures

One of the common features of most modern programming languages is the ability to do recursion. *Recursion* is the ability of functions or procedures to call themselves. Though not always mentioned, this ability is also available in SQL Query.

One of the classic programming problems is sorting. Sorting of rows is simple with SQL, but suppose you have a string field that contains comma separated values (CSV) that you need to sort; then SELECT with ORDER BY is not much use.

The requirement to sort the value in a file might arise if you are maintaining metadata. This is especially common for web back-end databases. Suppose you need to maintain a list of column numbers that have been changed by the front end. The first step is to find a way to number your columns. The best way is to use the same order as defined in the database.

Listing D-7

```
DECLARE @TableName varchar(50)

SET @TableName = 'Employees'

SELECT name FROM syscolumns
WHERE id = OBJECT_ID(@TableName)
ORDER BY colorder
```

The script in Listing D-7 returns the column names for the Employees table. The result is shown in Figure D-2.

	name
1	EmployeeID
2	LastName
3	FirstName
4	Title
5	TitleOfCourtesy
6	BirthDate
7	HireDate
8	Address
9	City
10	Region
11	PostalCode
12	Country
13	HomePhone
14	Extension
15	Photo
16	Notes
17	ReportsTo
18	PhotoPath

Figure D-2: The columns of the Employees table

Going back to our original CSV problem, suppose that a user changes the value in City. Our CSV will contain one value, “9.” If the user then changes Notes, HomePhone, Title, and Address, the final CSV will be “9, 16, 13, 4, 8.” Suppose now that CSV needs to be sorted so that it becomes “4, 8, 9, 13, 16,” and for the purpose of this exercise, let’s assume that you need this done in a stored procedure. You will have to implement some sort of storing routine. One of the most common sorting algorithms is called Quick Sort.

The Quick Sort Algorithm

The pseudocode for Quick Sort is fairly simple.

```

Start Sort
Get First Value in List
If there is other value then put all other values less
than current value in <Left List> and all values more than
current value in <Right List>; otherwise end the sort
Sort <Left List>      /* this is a recursive call*/
Sort <Right List>     /* this is a recursive call*/
Sorted list = Left List, Value, Right List
END Sort

```

The challenge now is in implementing the algorithm in SQL. We will call the stored procedure to do a Quick Sort of a CSV list `sp_QuickSort`.

The `sp_QuickSort` Stored Procedure

The first step is to make sure that if the stored procedure exists, we delete it so that we can create a new one. We also create the stored procedure but with no logic inside. This is not required, but it prevents getting warning messages when we apply the script with recursive calls.

Listing D-8a

```

IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'sp_QuickSort' AND type = 'P')
  DROP PROCEDURE sp_QuickSort
GO

create procedure sp_QuickSort
  @@CSVList varchar(256) = '' OUTPUT
AS
/*Create PROCEDURE first to allow recursion call to
  compile without warning*/
GO

```

Now that we have the blank stored procedure, the next step is to alter it and add the logic. We begin by declaring the variable we will need.

Listing D-8b

```

ALTER procedure sp_QuickSort
/*****
Procedure:   sp_sp_QuickSort
Date:       15-July-2002
Author:     Ryan N. Payet
Description: Quicksort a CSV list, also remove duplicate
            entry

Version History
-----
0.1, 15-July-2002, Ryan N. Payet
        Procedure Created

*****/
@@CSVList varchar(1000) = '' OUTPUT
AS
BEGIN

    IF @@CSVList = ''
        GOTO END_SORT

    /*Declare string variables to store list*/
    DECLARE @Left_List varchar(1000)
    DECLARE @Right_List varchar(1000)

    /*Declare string variables to
       store a single item in list*/
    DECLARE @strItem varchar(20)
    DECLARE @StrOneCSV varchar(20)

    /*Since items in list are numbers
       we need to covert string to numbers
       before we can do comparison
       First we declare integer variable to hold
       number for comparison*/
    DECLARE @intItem int

    /*Declare variables used to process list*/
    DECLARE @start int
    DECLARE @end int
    DECLARE @length int
    /*Set both lists to empty string*/

```

```
SET @Left_List = ''
SET @Right_List = ''
```

Now that we have declared our variable, we move on to getting the first item in the list. If we do not find a “,” (comma) character in the list, we can safely assume that there is no value in the list (empty list) or there is only one value in the list. In either case, there is no need to sort the list. If we find that the list has more than one item, then we get the first item in the list.

Listing D-8c

```
/*Start processing CSV so as to get first item*/
SET @start = 1

/*Find position of first "," */
SET @end = CHARINDEX(',', @@CSVList, @start)

/*if there is no "," then no need to sort.*/
IF @end = 0
    GOTO END_SORT

/*Get string length of first number*/
SET @length = @end - @start
IF @length = 0
    BEGIN
        SET @length = 1
        SET @end = 0
    END

/*Get first Value*/
SET @strItem = substring( @@CSVList , @start, @length )
SET @intItem = cast(@strItem as integer)
SET @start = @end + 1
SET @end = CHARINDEX(',', @@CSVList , @start)

/*Get other values in list*/
```

Now that we know the value of the first item, we can find the other items and build our right list of items greater than current value and left list for items less than current value.

► **Note:** Using only “greater than” (>) and “less than” (<) also removes duplicates in the list. To keep duplicates you must either use “greater than or equal to” (>=) on right list or use “less than or equal to” (<=) on left list.

Listing D-8d

```

WHILE (@end <> 0)
BEGIN
    SET @length = @end - @start

    SET @StrOneCSV = substring( @CSVList , @start,
        @length )
    SET @start = @end + 1
    SET @end = CHARINDEX(',', @CSVList , @start)

    /*If greater than first value,
    then add to Right_List
    If less than first value,
    add to Left_List*/
    IF CAST(@StrOneCSV as integer) > @intItem
        SET @Right_List = @StrOneCSV + ',' + @Right_List
    ELSE IF CAST(@StrOneCSV as integer) < @intItem
        SET @Left_List = @StrOneCSV + ',' + @Left_List

END

/*Process last item in list*/
SET @length = len( @CSVList )
SET @StrOneCSV = substring( @CSVList , @start,
    @length )
IF CAST(@StrOneCSV as integer) > @intItem
    SET @Right_List = @StrOneCSV + ',' + @Right_List
ELSE IF CAST(@StrOneCSV as integer) < @intItem
    SET @Left_List = @StrOneCSV + ',' + @Left_List

/*Clean lists of trailing "," character*/
SET @length = len(@Left_List) - 1
if @length > 0
    SET @Left_List = substring( @Left_List , 1, @length)

```

```

SET @length = len(@Right_List) - 1
if @length > 0
    SET @Right_List = substring( @Right_List , 1, @length)

```

All that remains is for us to sort the right list and left list using recursive calls to `sp_QuickSort`. Once we are done, we can rebuild the list.

Listing D-8e

```

/*quicksort front of list: <Left_List>*/
EXECUTE sp_QuickSort @Left_List OUTPUT

/*quicksort back of list: <Right_List>*/
EXECUTE sp_QuickSort @Right_List OUTPUT

/*Rebuild sorted list*/
if len(@Left_List) > 0
    SET @@CSVList = @Left_List + ',' + @strItem
ELSE
    SET @@CSVList = @strItem

if len(@Right_List) > 0
    SET @@CSVList = @@CSVList + ',' + @Right_List

END_SORT:
END

GO

```

Recursion is not really that difficult, once you master the basic algorithms. To simplify your SQL scripts, you should avoid having to do recursion logic. However, in certain circumstances where you do not have a choice, recursion can be very useful.

Let's now move to something more fun that can save a lot of time when used properly.

Dynamic Queries

Dynamic queries are queries that are non-deterministic. In other words, the actual query run, or the number of columns returned, are not known at design time but are determined by parameters at run time.

This is made possible by the ability of the EXECUTE statement and a system stored procedure called sp_executesql to execute scripts from strings.

For example, you can get the same result from

```
SELECT * FROM [Order Details]
```

by running the following

```
EXECUTE ('SELECT * FROM [Order Details]')
```

The difference is that with the EXECUTE statement, the script is in a string that we can programmatically manipulate. Consider the following:

```
DECLARE @SQLSCRIPT as nvarchar(1000)
DECLARE @TableName as nvarchar(100)

SET @SQLSCRIPT = 'SELECT * FROM '
SET @TableName = '[Order Details]'
```



```
EXECUTE (@SQLSCRIPT + @TableName)
```

► **Note:** Spaces in the string that you use are important.

By simply changing the value of @TableName to Orders, we can do a select on the Orders table.

```
DECLARE @SQLSCRIPT as nvarchar(1000)
DECLARE @TableName as nvarchar(100)

SET @SQLSCRIPT = 'SELECT * FROM '
SET @TableName = 'Orders'
```



```
EXECUTE (@SQLSCRIPT + @TableName)
```

This is all very fine and great, but the EXECUTE statement does have certain limitations. One of the major limitations is that the script in use cannot return any value. To overcome this, you have to use `sp_executesql`.

sp_executesql

`sp_executesql` has a simple syntax:

```
sp_executesql <Script>, [ <Parameter list> , Parameter1,
n...]
```

- `<Script>`: A string that contains the script you want to execute
- `<Parameter list>`: A string that contains the declaration of parameters used in `<Script>`
- `Parameter1, n...`: The parameters you want to pass in `<Script>` and/or the return value from `<Script>`

Consider the following situation. Let's assume that you need a stored procedure to return the rows for a given table name, a column name, and a search value. Let's call this stored procedure `sp_getRows`. The script for `sp_getRows` is shown in Listings D-9a to D-9c.

The first step, as usual, is to declare the stored procedure and variable required.

Listing D-9a

```
if exists
  (select * from dbo.sysobjects
   where id = object_id(N'sp_GetRows') and
   OBJECTPROPERTY(id, N'IsProcedure') = 1)
drop procedure sp_GetRows
GO

create procedure sp_GetRows
/*****
Procedure:   sp_UpdateContactInfo
Date:       16-July-2002
```

Author: Ryan N Payet
 Description: Return rows for the given table and search criteria

Version History

 0.1, 16-July-2002, Ryan N Payet:
 Procedure Created

```

*****/
@TableName nvarchar(100),
@ColumnName nvarchar(100) = '1',
@Value sql_variant = '1',
@Operator nvarchar (15) = '='
AS
BEGIN
  /*Declare variable to hold script*/
  DECLARE @@SQLScript as nvarchar(4000)
  DECLARE @@SQLPARAMETER as nvarchar(1000)

  /*Declare variable to manage errors*/
  DECLARE @@ERRORMSG as varchar(256)
  DECLARE @Error_Code as integer
  
```

 **Note:** `SQL_variant` is a data type that can store values of any SQL Server data types with the exception of text, `ntext`, timestamp, image, and `SQL_variant`.

We must make sure that the table name and column name given are valid.

Listing D-9b

```

/*Check if table exist*/
if NOT exists
  (SELECT * FROM dbo.sysobjects
   WHERE id = object_id(@TableName)
   and OBJECTPROPERTY(id, N'IsUserTable') = 1)
BEGIN
  SET @@ERRORMSG = 'Table "' + @TableName
  + '" does not exist in database'
  RAISERROR ( @@ERRORMSG , 16, 1)
  RETURN 1
END
  
```

```

/*If you base a number as column name then
do not check for existence of column.
This allows it to return all rows
by passing a column 1 and a value 1
*/
IF ISNUMERIC(@ColumnName) = 0
BEGIN
  IF NOT exists
    ( SELECT * FROM syscolumns
      WHERE (id = OBJECT_ID(@TableName))
        AND UPPER([name]) = UPPER(@ColumnName))
  BEGIN
    SET @@ERRORMSG = '
      + 'Column "' + @ColumnName
      + '" does not exist in table "'
      + @TableName + "'"
    RAISERROR (@@ERRORMSG , 16, 1)
    RETURN 1
  END /*IF NOT exists */
END /*IF ISNUMERIC(@ColumnName) = 0*/

```

We can then build our script. The point to note here is that @Value is not a variable but a string literal in the @@SQLScript variable. We also get the value of @@ERROR right after the SELECT in the script. We build a parameter list to be used with sp_executesql in @@SQLPARAMETER and when it is passed in sp_executesql. @error_code is defined as an OUTPUT type. This is necessary so that we can get a value from the script passed in sp_executesql. Also, the parameters must be used in the same order that they are defined in @@SQLPARAMETER.

Listing D-9c

```

/*Build Script*/
SET @@SQLScript = '
  + 'SELECT * FROM ' + @TableName + Char(13)
  + 'WHERE ' + @ColumnName
  + ' ' + @Operator +
  + ' @Value' + Char(13)
  + 'SET @Error_Code = @@ERROR'

```

```

/*Build parameter for script*/
SET @@SQLPARAMETER = '@Value sql_variant,'
    + ' @error_code integer OUTPUT'

/*This so that we know the script
    executed properly and set error_code to
    0 if there is no error*/
SET @error_code = 1

EXECUTE sp_executesql @@SQLScript,
    @@SQLPARAMETER ,
    @Value,
    @error_code OUTPUT

IF @error_code <> 0
BEGIN
    SET @@ERRORMSG = 'Error in script'
    RAISERROR (@@ERRORMSG , 16, 1)
    RETURN 1
END

RETURN 0

END
GO

```

 **Note:** CHAR(13) is ASCII character 13 (carriage return) and CHAR(39) is single quote character, useful when writing strings within a string.

To test the stored procedure, try the following script:

```
EXECUTE sp_GetRows 'Orders', 'EmployeeID', 6
```

With a lot of practice and creative programming, you can do a great deal more with dynamic scripts. One of the down sides of dynamic scripting, apart from the obvious complexity, is performance. Since the real SQL logic of the script is in string variables, you lose the benefit of executing plans and optimization by the database engine. This is not noticeable if the generated script is simple, but it can be an issue if the generated script is very complex.

Stored Procedure Generator

The idea of a stored procedure generator is to help the developer reduce the number of scripts that need to be written by using a stored procedure to create other stored procedures based on a template. This is only possible if the logic for all the procedures is the same with the exception of table names and column names. It also helps if there is a specific naming convention used.

For example, let's assume we have a database with more than 100 tables and we need a stored procedure in the form of [sp_getall <tablename>] that returns all rows for that table. For example, if there is a table called [Order Details], you need to create a stored procedure called [sp_getall Order Details].

You could create all of the stored procedures one by one, but this can be very tedious, so let's create it using dynamic script instead.

First let's create our stored procedure generator. We'll call it [sp_generate getall].

Listing D-10

```

if exists
  (select * from dbo.sysobjects
   where id = object_id(N'[sp_generate getall]') and
        OBJECTPROPERTY(id, N'IsProcedure') = 1)
drop procedure [sp_generate getall]
GO

create procedure [sp_generate getall]
/*****
Procedure:   sp_generate getall
Date:       16-July-2002
Author:     Ryan N Payet
Description: Generate stored procedure for given table

Version History
-----

```

```

0.1, 16-July-2002, Ryan N Payet:
    Procedure Created
*****/
    @TableName nvarchar(100)
AS
BEGIN

    /*Declare variable to hold script*/
    DECLARE @@SQLScript as nvarchar(4000)

    /*Declare variable to manage errors*/
    DECLARE @@ERRORMSG as varchar(256)

    /*Check if table exist*/
    if NOT exists
        (SELECT * FROM dbo.sysobjects
         WHERE id = object_id(@TableName)
         and OBJECTPROPERTY(id, N'IsUserTable') = 1)
    BEGIN
        SET @@ERRORMSG = 'Table "' + @TableName
            + '" does not exist in database'
        RAISERROR ( @@ERRORMSG , 16, 1)
        RETURN 1
    END

    /*Build script to drop stored procedure*/
    SET @@SQLScript = 'if exists' + char(13)
        + ' (select * from dbo.sysobjects' + char(13)
        + ' where id = object_id(N' + char(39)+ '[sp_getall '
        + @TableName + ']' + char(39)+ ') and ' + char(13)
        + ' OBJECTPROPERTY(id, N' + char(39) + 'IsProcedure'
        + char(39)+ ') = 1)' + char(13)
        + 'drop procedure [sp_getall ' + @TableName + ']'
        + char(13)

    /*Run script to drop stored procedure*/
    EXECUTE ( @@SQLScript )

    /*Build script to create stored procedure*/
    SET @@SQLScript = ''
        + 'create procedure [sp_getall ' + @TableName + ']'
        + char(13)
        + '/*****'
        + char(13)
        + 'Procedure:      [sp_getall ' + @TableName + ']'

```

```

+ char(13)
+ 'Date:      '+ CONVERT(varchar(20),GETDATE())
+ char(13)
+ 'Author:    Ryan N Payet (from Auto generator)'
+ char(13)
+ 'Description:  ' + char(13)
+ ' Return all rows for ' + char(13)
+ '*****/'
+ char(13)
+ 'AS ' + char(13)
+ 'BEGIN ' + char(13)
+ ' SELECT * FROM [' + @TableName + ']' + char(13)
+ 'END' + char(13)

/*Run script to create stored procedure*/
EXECUTE ( @@SQLScript )

END
GO

```

[sp_generate getall] is a simple stored procedure that, through the use of dynamic script, creates other stored procedures. The generated stored procedures also include comments and the date they are generated.



Tip: The maximum size nvarchar can be is 8,000, which means it holds only 4,000 characters. If your dynamic script is more than 4,000 characters long, use more than one variable to hold the scripts and split the scripts among each. When you execute the script, you can do the following:

```
EXECUTE (@@SQLScript1 + @@SQLScript2 + @@SQLScript3)
```

Now that the stored procedure generator is ready, you can move on and write a line that will generate a stored procedure for each table. For example:

```

...
EXECUTE [sp_generate getall] 'Orders'
EXECUTE [sp_generate getall] 'Order Details'
...

```

Or you can be more clever and write a script that will do it for you through the use of a cursor:

Listing D-11

```

/*
  Create script to generate stored procedure
  for all table in database
*/

/*Declare variable to hold table name*/
DECLARE @tableName as nvarchar(100)

/*Declare cursor for Loop though table name*/
DECLARE tablename_Cursor CURSOR FOR
  SELECT name FROM dbo.sysobjects
  WHERE OBJECTPROPERTY(id, N'IsUserTable') = 1

OPEN tablename_Cursor

/*Get the first table name*/
FETCH NEXT FROM tablename_Cursor
INTO @tableName

/*LOOP until end of cursor*/
WHILE @@FETCH_STATUS = 0
BEGIN
  /*GENERATE stored procedure*/
  EXECUTE [sp_generate getall] @tableName

  /*Get next table name*/
  FETCH NEXT FROM tablename_Cursor
  INTO @tableName
END

/*CLEAN up*/
CLOSE tablename_Cursor
DEALLOCATE tablename_Cursor

GO

```

This technique is practical if you only have a few tables and stored procedures to write, but it is extremely useful if you need to write a large number of stored procedures with similar logic.

GROUP BY with CUBE and ROLLUP

GROUP BY is used in combination with aggregate functions to calculate statistical values. The columns referred to by GROUP BY are sometimes called *dimension columns*. CUBE and ROLLUP are used to summarize the value further to show subtotals and totals. The differences between CUBE and ROLLUP are:

- CUBE generates subtotals for all combinations of values in dimension columns.
- ROLLUP generates subtotals for a hierarchy of values in dimension columns.

For example, if you want to find out the sale value of Orders, then you would run the following query:

Listing D-12a

```
SELECT SUM( [Order Details].UnitPrice *
            [Order Details].Quantity *
            (1 - [Order Details].Discount)
        ) as [Sale Value]
FROM   [Order Details]
```

This is a simple query and will return just one value. Now let's do something more complicated and try to find the sale value for each product.

Listing D-12b

```
SELECT Products.ProductName,
       SUM( [Order Details].UnitPrice *
           [Order Details].Quantity *
           (1 - [Order Details].Discount)
       ) as [Sale Value]
FROM   [Order Details]
       INNER JOIN Products
       ON [Order Details].ProductID = Products.ProductID
GROUP BY Products.ProductName
ORDER BY Products.ProductName
```

The result will show the total for ProductName. We are assuming that ProductName is unique for each product. Figure D-3 shows a small part of a sample result set.

	ProductName	Sale Value
1	Alice Mutton	32698.379981994629
2	Aniseed Syrup	3044.0
3	Boston Crab Meat	17910.629892349243
4	Camembert Pierrot	46825.480133056641
5	Carnarvon Tigers	29171.875
6	Chai	12893.100059509277
7	Chang	16379.96004486084
8	Chartreuse verte	12294.539949417114
9	Chef Anton's Cajun S...	8567.8999938964844
10	Chef Anton's Gumbo Mix	5347.2000045776367
11	Chocolate	1368.7125244140625
12	Côte de Blaye	141396.73522949219
13	Escargots de Bourgogne	5881.675048828125
14	Filo Mix	3232.9499950408936
15	Flotemysost	19551.025001525879
16	Geitost	1648.125
17	Genen Shouvu	1784.8249969482422

Figure D-3: Sample result set for sale value of products using GROUP BY

The result, however, does not show total sale value. To do this, you need to use ROLLUP or CUBE. The result is the same for both because there is only one non-aggregate column.

Listing D-12c

```

SELECT Products.ProductName,
       SUM( [Order Details].UnitPrice *
           [Order Details].Quantity *
           (1 - [Order Details].Discount)
         ) as [Sale Value]
FROM   [Order Details]
       INNER JOIN Products
         ON [Order Details].ProductID = Products.ProductID
GROUP BY Products.ProductName WITH ROLLUP
ORDER BY Products.ProductName

```

The result produces an extra row with a value of null for ProductName and the grand total for sale value. See Figure D-4.

	ProductName	Sale Value
1	NULL	1266331.0396184921
2	Alice Mutton	32698.379981994629
3	Aniseed Syrup	3044.0
4	Boston Crab Meat	17910.629892349243
5	Camembert Pierrot	46825.480133056641
6	Carnarvon Tigers	29171.875
7	Chai	12893.100059509277
8	Chang	16379.96004486084
9	Chartreuse verte	12294.539949417114
10	Chef Anton's Cajun Seasoning	8567.8999938964844

Figure D-4: Extra row produced by GROUP BY with ROLLUP

With CUBE and ROLLUP, null means all values for that particular column. We can modify the query so that it reflects that fact.

Listing D-12d

```

SELECT CASE WHEN
    (GROUPING(Products.ProductName) = 1)
    THEN '<-- ALL -->'
    ELSE
        ISNULL(Products.ProductName, 'UNKNOWN')
    END AS [Product Name],
    SUM( [Order Details].UnitPrice *
        [Order Details].Quantity *
        (1 - [Order Details].Discount)
        ) as [Sale Value]
FROM [Order Details]
INNER JOIN Products
    ON [Order Details].ProductID = Products.ProductID
GROUP BY Products.ProductName WITH ROLLUP
ORDER BY [Product Name]

```

Now null is replaced by <-- ALL -->.

► **Note:** GROUPING (<Column Name>) is an aggregate function that returns a value of 1 when the row is added by either the CUBE or ROLLUP operator or 0 when the row is not the result of CUBE or ROLLUP.

Let's expand the example to include "Employee Name." Employee Name is a computed field made up of FirstName and LastName. To match to unique employees we use the Employees table primary key, EmployeeID, in the GROUP BY. GROUP BY restricts the column that you can use in the SELECT section without the use of an aggregate function to only those referred to in the GROUP BY section of the script. To overcome this limitation, we can use the aggregate function MIN or MAX (minimum value and maximum value, respectively) on FirstName and LastName. Since for any given EmployeeID, the FirstName and LastName are the same; then the actual value for FirstName and LastName is returned. We can also include a special field that we can use for sorting so that all the totals are together. Obviously, the one with the biggest group value is the grand total.

Listing D-12e

```
SELECT CASE WHEN
    (GROUPING(Employees.EmployeeID) = 1)
    THEN '<-- ALL -->'
    ELSE
        ISNULL( max( Employees.FirstName) + ' ' +
                max(Employees.LastName),
                'UNKNOWN')
    END as [Employee Name] ,
    CASE WHEN
        (GROUPING(Products.ProductName) = 1)
    THEN '<-- ALL -->'
    ELSE
        ISNULL(Products.ProductName, 'UNKNOWN')
    END AS [Product Name],
    SUM( [Order Details].UnitPrice *
        [Order Details].Quantity *
```

```

        (1 - [Order Details].Discount)
    ) as [Sale Value],
    GROUPING(Employees.EmployeeID)
+ GROUPING(ProductName) as [Group Value]
FROM Employees
INNER JOIN Orders
    ON Employees.EmployeeID = Orders.EmployeeID
INNER JOIN
[Order Details]
    ON Orders.OrderID = [Order Details].OrderID
INNER JOIN Products
    ON [Order Details].ProductID = Products.ProductID
GROUP BY Employees.EmployeeID, Products.ProductName WITH
ROLLUP
ORDER BY [Group Value],
[Employee Name],
Products.ProductName

```

The additional rows generated by ROLLUP are shown in Figure D-5.

	Employee Name	Product Name	Sale Value	Group
587	Steven Buchanan	Uncle Bob's Organic Dried Pears	1350.0	0
588	Steven Buchanan	Veggie-spread	263.39999389648437	0
589	Steven Buchanan	Winners gute Semmelknödel	473.8125	0
590	Steven Buchanan	Zaanse koeken	356.25	0
591	Andrew Fuller	<-- ALL -->	166537.75497817993	1
592	Anne Dodsworth	<-- ALL -->	77308.066709518433	1
593	Janet Leverling	<-- ALL -->	202812.84279346466	1
594	Laura Callahan	<-- ALL -->	127027.27770423889	1
595	Margaret Peacock	<-- ALL -->	232890.84594726562	1
596	Michael Suyama	<-- ALL -->	74203.129243850708	1
597	Nancy Davolio	<-- ALL -->	192107.60432052612	1
598	Robert King	<-- ALL -->	124568.23538208008	1
599	Steven Buchanan	<-- ALL -->	68875.282539367676	1
600	<-- ALL -->	<-- ALL -->	1266331.0396184921	2

Figure D-5: Additional rows generated by ROLLUP

As you can see in the figure above, the sample data I used resulted in 600 rows being generated. ROLLUP finds all combinations of Employee Name against Product Name. If we use CUBE instead, more additional rows will be generated because CUBE will generate all possible combinations of Employee Name against Product Name and also all possible combinations of Product Name against Employee Name.

Employee Name	Product Name	Sale Value	Group
659 <-- ALL -->	Thüringer Rostbratwurst	80368.671997070313	1
660 <-- ALL -->	Tofu	7991.4900035858154	1
661 <-- ALL -->	Tourtière	4728.2375335693359	1
662 <-- ALL -->	Tunnbröd	4601.699969482422	1
663 <-- ALL -->	Uncle Bob's Organic Dried Pears	22194.299987792969	1
664 <-- ALL -->	Valkoinen suklaa	3437.6875	1
665 <-- ALL -->	Vegie-spread	16701.095001220703	1
666 <-- ALL -->	Wimmers gute Semmelknödel	21957.967346191406	1
667 <-- ALL -->	Zaanse koeken	3958.0799865722656	1
668 Andrew Fuller	<-- ALL -->	166537.75497817993	1
669 Anne Dodsworth	<-- ALL -->	77308.066709518433	1
670 Janet Leverling	<-- ALL -->	202812.84279346466	1
671 Laura Callahan	<-- ALL -->	127027.27770423889	1
672 Margaret Peacock	<-- ALL -->	232890.84594726562	1
673 Michael Suyama	<-- ALL -->	74203.129243850708	1
674 Nancy Davolio	<-- ALL -->	192107.60432052612	1
675 Robert King	<-- ALL -->	124568.23538208008	1
676 Steven Buchanan	<-- ALL -->	68875.282539367676	1
677 <-- ALL -->	<-- ALL -->	1266331.0396184921	2

Figure D-6: Some of the additional rows generated by CUBE

As you can see in Figure D-6, with the sample data I used, CUBE generated 677 rows compared to the 600 rows by ROLLUP.

Since CUBE generated a lot of data and might be difficult to read, the best way to use CUBE is to make it into a view and then use the view to see only the part of the data that you are interested in.

Listing D-13a

```

if exists
    (select * from sysobjects where id = object_id(N'[CUBE
      EmployeeProduct]')
      and OBJECTPROPERTY(id, N'IsView') = 1)
drop view [CUBE EmployeeProduct]
GO

CREATE VIEW [CUBE EmployeeProduct]
AS
SELECT CASE WHEN
    (GROUPING(Employees.EmployeeID) = 1)
    THEN '<-- ALL -->'
    ELSE

```

```

        ISNULL( max( Employees.FirstName) + ' ' +
                max(Employees.LastName),
                'UNKNOWN')
    END as [Employee Name] ,
CASE WHEN
    (GROUPING(Products.ProductName) = 1)
THEN '<-- ALL -->'
ELSE
    ISNULL(Products.ProductName, 'UNKNOWN')
END AS [Product Name],
SUM( [Order Details].UnitPrice *
      [Order Details].Quantity *
      (1 - [Order Details].Discount)
    ) as [Sale Value],
GROUPING(Employees.EmployeeID)
+ GROUPING(ProductName) as [Group Value]
FROM Employees
INNER JOIN Orders
    ON Employees.EmployeeID = Orders.EmployeeID
INNER JOIN
[Order Details]
    ON Orders.OrderID = [Order Details].OrderID
INNER JOIN Products
    ON [Order Details].ProductID = Products.ProductID
GROUP BY Employees.EmployeeID, Products.ProductName WITH
CUBE

GO

```

Once you have created the view, you can reduce the amount of data returned.

Listing D-13b

```

SELECT * FROM [CUBE EmployeeProduct]
WHERE [Product Name] = '<-- ALL -->'
ORDER BY [Group Value],
         [Employee Name]

```

The result is shown in Figure D-7.

	Employee Name	Product Name	Sale Value	Group Value
1	Andrew Fuller	<-- ALL -->	166537.75497817993	1
2	Anne Dodsworth	<-- ALL -->	77308.066709518433	1
3	Janet Leverling	<-- ALL -->	202812.84279346466	1
4	Laura Callahan	<-- ALL -->	127027.27770423889	1
5	Margaret Peacock	<-- ALL -->	232890.84594726562	1
6	Michael Suyama	<-- ALL -->	74203.129243850708	1
7	Nancy Davolio	<-- ALL -->	192107.60432052612	1
8	Robert King	<-- ALL -->	124568.23538208008	1
9	Steven Buchanan	<-- ALL -->	68875.282539367676	1
10	<-- ALL -->	<-- ALL -->	1266331.0396184921	2

Figure D-7: The return result from the view [CUBE EmployeeProduct]

Server Cross Tabulations with SQL

Cross tabulation (cross-tab or X-tab for short) is a statistical report where data is displayed in a way that allows for easy comparison. The displayed data is usually aggregated, denormalized, and displayed within a matrix table where the headings of certain rows are data themselves.

Consider this table, which shows some data about certain items from a small electronics store.

Name	Status	Amount
VCR player	Damaged	1
VCR player	Sold	10
Monitor	Damaged	3
Monitor	Sold	3
Mouse	Damaged	0
Mouse	Sold	1
Keyboard	Damaged	2
Keyboard	Sold	1

The table on the following page shows another way to display the same information.

Name	Sold	Damaged	Total
VCR player	10	1	11
Monitor	3	3	6
Mouse	1	0	1
Keyboard	1	2	3
Total	15	6	21

The new table formatted as a cross-tab now displays the data in a more compact manner that allows for easier comparison. Notice that the Status column, which contains two discrete values (Sold, Damaged), has been replaced by the Sold and Damaged columns. In other words, the value from Status has been converted to columns. If Status had more values, the cross-tab would generate a column for each additional value. There is also a column and a row to show the total for each.

Many front-end tools provide advanced cross-tab capabilities. PivotTable in Excel is a good example. A problem, however, arises when you have a huge amount of data that you need to transfer. This tends to slow down the front end. The way around this is to do the cross-tab, or some of it, on the server. Very complex cross-tabs and statistical analysis can be achieved through the use of expensive statistical tools, such as online analytical processing (OLAP) tools available with the Enterprise Edition of Microsoft SQL Server 2000. However, this might be too expensive or overkill for what you need.

Let's start with a simple example. If you look at the Employees table, you will notice that there are two distinct values for the Country field. These are "USA" and "UK." Suppose that for each job title (Employees.Title), we want to know how many employees are from each country. This would be a simple matter of using COUNT and GROUP BY.

Listing D-14a

```

SELECT Employees.Title,
       Employees.Country,
       COUNT(1) as total
FROM Employees
GROUP BY Employees.Title, Employees.Country
ORDER BY Employees.Title

```

The result is shown in Figure D-8:

	Title	Country	total
1	Inside Sales Coordinator	USA	1
2	Sales Manager	UK	1
3	Sales Representative	UK	3
4	Sales Representative	USA	3
5	Vice President, Sales	USA	1

Figure D-8: Result of counting employees for each Title and Country

Though the result list is small, imagine for a minute that there are many more employees and many more titles; then the result set would be bigger and difficult to read. To overcome this, you can put USA and UK under their own column heading so that you end up with:

Title	USA	UK	Total
Inside Sales Coordinator	1	0	1
Sales Manager	0	1	1
Sales Representative	3	3	6
Vice President, Sales	1	0	1
<-- ALL -->	5	4	9

To work the query out, it helps if you consider how you would do it manually.

- For USA: You will only count a row if the country is USA.

- For UK: You will only count a row if the country is UK.

Let's now put this into a SQL query.

Listing D-14b

```
SELECT Employees.Title,
       SUM (CASE WHEN Employees.Country = 'USA' THEN 1
             ELSE 0 END) as USA,
       SUM (CASE WHEN Employees.Country = 'UK' THEN 1
             ELSE 0 END) as UK,
       COUNT(1) as total
FROM Employees
GROUP BY Employees.Title
ORDER BY Employees.Title
```

Since Country is no longer a column in the SELECT but has been split into two columns, USA and UK, we must remove it from the GROUP BY clause. This query, however, does not have our grand total (<-- ALL -->). To generate it, we need to make use of ROLLUP, like we learned in the previous section. We also have to generate an extra column [Group No] to help with the ordering.

```
SELECT GROUPING(Employees.Title) as [Group No],
       CASE WHEN
         (GROUPING(Employees.Title) = 1)
       THEN '<-- ALL -->'
       ELSE
         ISNULL(Employees.Title, 'UNKNOWN')
       END AS Title,
       SUM (CASE WHEN Employees.Country = 'USA' THEN 1
             ELSE 0 END) as USA,
       SUM (CASE WHEN Employees.Country = 'UK' THEN 1
             ELSE 0 END) as UK,
       COUNT(1) as Total
FROM Employees
GROUP BY Employees.Title WITH ROLLUP
ORDER BY [Group no], Employees.Title
```

The result is shown in Figure D-9.

	Group No	Title	USA	UK	Total
1	0	Inside Sales Coordinator	1	0	1
2	0	Sales Manager	0	1	1
3	0	Sales Representative	3	3	6
4	0	Vice President, Sales	1	0	1
5	1	<-- ALL -->	5	4	9

Figure D-9: Simple cross-tab for employees

There is one problem, however. If we get an employee from another country, say for example, Seychelles (SEZ), we will have to recreate the script with an extra column. What if there are 20 more countries and 50 more different titles? It is not viable to create the columns one by one. There is something that we learned earlier that we can use: dynamic queries.

Cross-tab with Dynamic Queries

To demonstrate the power of dynamic queries, GROUP BY, and cross-tabs, let's suppose we want to compare the sales value of every product against every employee to generate a product-employee cross-tab. We will do this through a stored procedure called [sp_Product_Employee_xtab].

The first design consideration is to choose which of the Product and Employee columns you want to split into separate columns. Since there are fewer employees than there are products, it is better to choose Employee so that we have fewer columns to build.

The SELECT Templates

The general template of the SELECT in this case is:

Listing D-15a

```
SELECT GROUPING(ProductName) as [Group No],
CASE WHEN (GROUPING(Products.ProductName) = 1)
THEN '<-- ALL -->'
ELSE
```

```

        ISNULL(Products.ProductName, 'UNKNOWN ')
    END AS [Product Name],
    <CASE WHEN for each Employee Name>
    SUM( [Order Details].UnitPrice *
        [Order Details].Quantity *
        (1 - [Order Details].Discount)
    ) as [Total Sale Value]
FROM Employees
    INNER JOIN Orders
        ON Employees.EmployeeID = Orders.EmployeeID
    INNER JOIN [Order Details]
        ON Orders.OrderID = [Order Details].OrderID
    INNER JOIN Products
        ON [Order Details].ProductID = Products.ProductID
GROUP BY Products.ProductName WITH ROLLUP
ORDER BY [Group No], Products.ProductName

```

The template for <CASE WHEN for each Employee Name> is shown in Listing D-15b.

Listing D-15b

```

SUM ( CASE WHEN Employees.EmployeeID = <@EmployeeID>
THEN ( [Order Details].UnitPrice *
        [Order Details].Quantity *
        (1 - [Order Details].Discount)
    )
ELSE
    0
END )as [<@EmployeeName>]

```

The templates can be generated into a string and the string executed using EXECUTE.

The sp_Product_Employee_xtab

Now that we know the template for the SELECT statement, let's go ahead and build sp_Product_Employee_xtab. First, we do the standard declarations.

Listing D-16a

```

/*****
Drop stored procedure if already exist *
*****/
if exists

```

```

(select * from sysobjects
 where id = object_id(N'sp_Product_Employee_xtab') and
  OBJECTPROPERTY(id, N'IsProcedure') = 1
)
DROP procedure sp_Product_Employee_xtab
GO

CREATE procedure sp_Product_Employee_xtab
/*****
Procedure:  sp_OrderAddProduct
Date:      17 July 2002
Author:    Ryan N. Payet
Description:
           Generate Cross-tab to compare Sale Value
           of every product against every Employee

Version History
-----
0.1, 17 July 2002, Ryan N. Payet:
      Procedure Created
*****/
AS
BEGIN

/*Declare Variable for script*/
DECLARE @@SQLTOPScript as nvarchar(4000)
DECLARE @@SQLCOLScript as nvarchar(4000)
DECLARE @@SQLBOTTOMScript as nvarchar(4000)

/*Declare Variable for employee*/
DECLARE @@ID as integer
DECLARE @@NAME as nvarchar(100)

```

Now we need to generate the <CASE WHEN for each Employee Name> part. To do this, we need to use a cursor that accesses the Employee table. Using the cursor, we can then loop through each row in the Employees table and generate the script.

Listing D-16b

```

/*Declare cursor for Loop though Employees table */
DECLARE Employee_Cursor CURSOR FOR
  SELECT EmployeeID,
         Employees.FirstName + ' ' + Employees.LastName as

```

```

        [Employee Name]
    FROM Employees ORDER BY [Employee Name]

OPEN Employee_Cursor

/*Get the first Employee*/
FETCH NEXT FROM Employee_Cursor
INTO @@ID,@@NAME
SET @@SQLCOLScript = ''
/*LOOP until end of cursor*/
WHILE @@FETCH_STATUS = 0
BEGIN
    /*Build Column list*/
    SET @@SQLCOLScript = @@SQLCOLScript + ',' + Char(13)
        + ' SUM ( CASE WHEN Employees.EmployeeID = '
        + CONVERT(nvarchar(10),@@ID)
        + ' THEN ( [Order Details].UnitPrice '
        + ' * [Order Details].Quantity * '
        + ' (1 - [Order Details].Discount) )'
        + ' ELSE 0 END)as [' + @@NAME + ']'
    /*Get next table name*/
    FETCH NEXT FROM Employee_Cursor
    INTO @@ID,@@NAME
END

/*CLEAN up CURSOR*/
CLOSE Employee_Cursor
DEALLOCATE Employee_Cursor

```

The final part is simple, and all that needs to be done is to generate the rest of the scripts, join them, and execute.

Listing D-16c

```

/*Declare top part of query*/
SET @@SQLTOPScript = ''
+ 'SELECT '
+ ' GROUPING(ProductName) as [Group No],' + Char(13)
+ ' CASE WHEN '
+ ' (GROUPING(Products.ProductName) = 1)'
+ ' THEN ' + CHAR(39) + '<-- ALL -->' + CHAR(39)
+ ' ELSE'
+ ' ISNULL(Products.ProductName, '

```

```

+ CHAR(39) + 'UNKNOWN ' + CHAR(39) + ') '
+ 'END AS [Product Name]'
/*Declare bottom part of query*/
SET @@SQLBOTTOMScript = ',' + Char(13)
+ ' SUM( [Order Details].UnitPrice *'
+ ' [Order Details].Quantity *'
+ ' (1 - [Order Details].Discount)'
+ ' ) as [Total Sale Value] ' + Char(13)
+ 'FROM Employees ' + Char(13)
+ ' INNER JOIN Orders ' + Char(13)
+ ' ON Employees.EmployeeID = Orders.EmployeeID '
+ Char(13)
+ ' INNER JOIN [Order Details] ' + Char(13)
+ ' ON Orders.OrderID = [Order Details].OrderID '
+ Char(13)
+ ' INNER JOIN Products' + Char(13)
+ ' ON [Order Details].ProductID
= Products.ProductID' + Char(13)
+ ' GROUP BY Products.ProductName WITH ROLLUP '
+ Char(13)
+ ' ORDER BY [Group No], Products.ProductName '
+ Char(13)

EXECUTE ( @@SQLTOPScript + @@SQLCOLScript
+ @@SQLBOTTOMScript)

END

GO

```

Now that the stored procedure is done, all we have to do is execute it to get our cross-tab.

```
EXECUTE sp_Product_Employee_xtab
```

A sample of the result is shown in Figure D-10.

The stored procedure returns a result that is formatted as a cross-tab. If the number of employees increases, the number of columns will automatically be adjusted to accommodate the new employees.

G.	Product Name	Andrew Fuller	Anne Dodsworth	Janet Leverling
57	0 Rössle Sauerkraut	1299.4800109863281	0.0	2827.199951171875
58	0 Sasquatch Ale	854.0	112.0	838.6000061035156
59	0 Schoggi Schokolade	5707.0	0.0	1755.0
60	0 Scottish Longbreads	255.0	0.0	1212.5
61	0 Singaporean Hokkien Fried Mee	2885.4000091552734	0.0	674.8000011444091
62	0 Sir Rodney's Marmalade	0.0	4252.5	6188.400024414062
63	0 Sir Rodney's Scones	686.0	370.0	1136.0
64	0 Sirop d'érable	4277.85009765625	114.0	2094.75
65	0 Spegesild	840.0	288.0	274.7999877929687
66	0 Steeleye Stout	2025.0	856.800048828125	524.6999969482421
67	0 Tarte au sucre	7392.9998397827148	0.0	9376.324920654296
68	0 Teatime Chocolate Biscuits	920.12999725341797	322.0	534.09998899373779
69	0 Thüringer Rostbratwurst	15845.119995117188	4208.8599853515625	12317.10516357421
70	0 Tofu	0.0	98.8125	571.9499969482421
71	0 Tourtière	273.08000564575195	237.5	1477.350021362304
72	0 Tunnbröd	241.19999694824219	0.0	468.0
73	0 Uncle Bob's Organic Dried Pears	3870.0	0.0	1800.0
74	0 Valkoinen suklaa	0.0	0.0	958.75
75	0 Vegie-spread	505.43997192382812	2386.800048828125	2677.375
76	0 Wimmers gute Semmelknödel	1602.6499633789062	997.5	3378.199951171875
77	0 Zaanse koeken	95.0	285.0	876.2799987792968
78	1 <-- ALL -->	166537.75497817993	77308.066709518433	202812.8427934646

Figure D-10: Sample result of running `sp_Product_Employee_xtab`

Summary

The techniques discussed in this appendix will go a long way in helping you build an efficient and feature-rich application. There are many ways that the same end result can be accomplished, but knowing different sets of techniques gives the developer additional tools so the correct tool can be chosen for the job. In the final analysis, it is all about choices.

Index

- AcceptChanges() method, 76
- AcceptChangesDuringFill property, 57
- ActiveX Data Object Multidimensional,
see ADO MD
- ADO,
 - architecture, 4-7
 - data types, 6-7
 - using JOIN in, 12-13
 - using with ADO .NET, 250-251
- ADO MD, 274-275
- ADO .NET,
 - architecture, 7-9
 - integration with XML, 133-134
 - using with ADO, 250-251
- Analysis Services, 262
 - architecture of, 270-271
 - installing, 262-265
 - setting up, 266-270
- application, protecting, 129-130
- application scope, 122
- architecture,
 - ADO, 4-7
 - ADO .NET, 7-9
 - differences between ADO and
ADO .NET, 3
- ASP, using with ASP .NET, 250
- ASP .NET applications, 106
 - data access in, 115-119
 - using with ASP, 250
- attributes, 136
- BeginInit() method, 76-77
- BeginTransaction() method, 24-25
- Boolean expressions, evaluating, 247
- Cancel() method, 33
- CaseSensitive property, 71
- casting, explicit, 248
- Catalog object, 276
- ChangeDatabase() method, 25
- classes, 302
 - creating, 304-307
 - defining property in, 124-125
 - implementing in VB .NET,
302-303
 - using public fields in, 123-124
- Clear() method, 77
- Clone() method, 78
- Close() method, 25-26
- COM, 249-250
 - using with .NET, 249-250
- Command object, 30
 - methods, 33-37
 - properties, 30-32
- CommandText property, 30
- CommandTimeout property, 30-31
- CommandType property, 31
- components, 121
 - designing, 122
 - implementing, 123-124
 - reasons for building, 121-122
- composite data types, 301
- concurrency, 119-120, 219
- Connection object, 20
 - methods, 24-28
 - properties, 21-24
- connection pooling, 125-127
- Connection property, 31-32
- connection string, defining, 181-184
- ConnectionString property, 21-22
- ConnectionTimeout property, 22
- console applications, 104
 - data access in, 104-105
- constraints, mapping, 166-169

Index

- container, 121
- Container property, 32, 71
- ContinueUpdateOnError property, 58
- Copy() method, 78-79
- CreateCommand() method, 26
- CreateObjRef() method, 39-40, 62
- CreateParameter() method, 33
- cross-tabs, 407
 - using, 407-411
 - using with dynamic queries, 411-412
- cross-tabulations, *see* cross-tabs
- cube, 273
- CUBE, 400
 - using, 401-406
- Cube object, 276-277
- CubeBrowser ActiveX control, using, 285-297
- data access,
 - improving, 125-128
 - in ASP .NET applications, 115-119
 - in console applications, 104-105
 - in Web Forms, 107-109
 - in Windows Forms applications, 103-104
 - in Windows Services applications, 105-106
- data binding, 100-102
 - reasons for using, 102-103
- data retrieval methods, 175
- data types, 6-7, 301
 - choosing, 127-128
 - for XML Schema, 141-142
- data update methods, 175, 210
- data warehousing, 128
- DataAdapter object, 54
 - initializing, 55-57
- Database property, 22
- DataColumn object, binding to, 100
- DataReader object, 37
 - methods, 39-53
 - properties, 37-39
- DataRelation object, 13-14
 - nested, 159-160
- DataRelationCollection property, 94-95
- DataSet object, 10, 68
 - adding extended property to, 92-93
 - binding to, 101
 - defining schema, 154-155
 - initializing, 10-12
 - methods, 76-92
 - organization of, 70-71
 - properties, 71-76
 - synchronizing with XmlDataDocument, 157-159
 - typed, 155-156
 - using, 69
 - using in XML Web Service, 112-115
 - using with XML, 144
 - using with XmlDataDocument, 156-157
- DataSetName property, 72
- DataSource property, 23
- DataTable object, 10, 95
 - binding to, 100-101
 - creating, 95-98
- DataTableCollection property, 93-94
- DataView object, binding to, 101
- DataViewManager object, binding to, 101
 - default properties, 247
- DefaultViewManager property, 72
- DELETE, using with WHERE clause, 381-383
- DeleteCommand property, 58
- Depth property, 37-38
- DesignMode property, 72
- DiffGram format, 111, 145-149
- dimension columns, 400
- Dimension object, 277
- Dispose() method, 26-27, 34, 62, 79
- Document Object Model, *see* DOM
- document type declaration, *see* DTD
- DOM, 134-135
 - loading XML documents in, 138-139
- DTD, 140

- dynamic queries, 391-392
 - using with cross tabs, 411-412
- encapsulation, 316
- EndInit() method, 79-80
- EnforceConstraints property, 73
- Enterprise Manager, using with views, 340-342
- Equals() method, 27, 40
- evaluation of Boolean expressions, 247
- ExecuteNonQuery() method, 34-35
- ExecuteReader() method, 35
- ExecuteScalar() method, 35-36
- ExecuteXmlReader() method, 36
- explicit casting, 248
- extended property,
 - adding to DataSet, 92-93
 - reading value of, 93
 - writing value to, 93
- ExtendedProperties property, 73, 92-93
- extensibility of XML, 132
- Extensible Markup Language, *see* XML
- Extensible Stylesheet Language Transformation, *see* XSLT
- FieldCount property, 38
- Fill() method, 63-64
- FillSchema() method, 64-65
- First Normal Form, 327-329
- FoodMart example, relational database of, 271-272
- form data binding, 100-102
- Get procedure, 313-314
- GetBoolean() method, 40-41
- GetByte() method, 41
- GetBytes() method, 41-42
- GetChanges() method, 80
- GetChar() method, 42
- GetChars() method, 43
- GetDataTypeName () method, 43-44
- GetDateTime() method, 44
- GetDecimal() method, 44-45
- GetDouble() method, 45
- GetFieldType() method, 45-46
- GetFloat() method, 46
- GetGuid() method, 46-47
- GetHashCode() method, 47
- GetInt16() method, 47-48
- GetInt32() method, 48
- GetInt64() method, 49
- GetName() method, 49
- GetOrdinal() method, 50
- GetSchemaTable() method, 50
- GetService() method, 80-81
- GetString() method, 50-51
- GetTimeSpan() method, 51
- GetType() method, 27, 36, 81
- GetValue() method, 51-52
- GetValues() method, 52
- GetXml() method, 152
- GetXML() method, 82
- GetXmlSchema() method, 82-83, 152
- GROUP BY, using, 400-407
- HasChanges() method, 83-84
- HasErrors property, 73
- Hierarchy object, 277
- IIS, 178
 - setting up, 178-179
- InferXmlSchema() method, 84-86
- inheritance, 319-320
 - implementing, 320-324
- InsertCommand property, 59
- Internet Information Server, *see* IIS
- IsClosed property, 38
- IsDBNull() method, 52-53
- Item property, 38
- JOIN statement, using, 12-13
- languages, differences between, 246-249
- libraries, in Visual Studio, 288-289
- Locale property, 74
- lookup table, 103
- MDX, 274
 - member, 308
 - implementing, 310-312

Index

- member variable, 308
 - Merge() method, 86-88
 - methods, 318
 - migrating,
 - ASP application to ASP .NET, 251-253
 - issues with, 257-258
 - steps for, 258-259
 - MissingMappingAction property, 59
 - MissingSchemaAction property, 60
 - monitoring, 128
 - MS SQL Server 2000, using with XML, 143-144
 - Multidimensional Expressions, *see* MDX
 - namespace, Web Service, 180-181
 - Namespace property, 74
 - .NET,
 - application models, 99
 - component, 121
 - conversion issues, 246-249
 - data providers, 9
 - data types, 6-7
 - nodes, 137-138
 - using with COM, 249-250
 - .NET Framework, integration with XML, 131-134
 - .NET InterOp services, 250
 - New() method, 55-57
 - NextResult() method, 53
 - nodes, 135-136
 - .NET, 137-138
 - normalization, 325
 - issues with, 332-333
 - process of, 326
 - Northwind example, functionality for, 174-175
 - object, 302
 - object orientation in VB .NET, 301-324
 - OLAP, 262
 - cubes, 273
 - OLAP database, 272
 - designing, 278-283
 - populating, 272
 - OLEDB Provider for OLAP, 273-274
 - OLTP database, 128
 - online analysis processing database, *see* OLAP database
 - online transaction processing database, *see* OLTP database
 - Open() method, 28
 - optimistic concurrency, 119
 - OrderProcessingWS example, 176
 - creating, 179-180
 - data retrieval methods in, 176-177
 - data update methods in, 209-210
 - DeleteOrderDetails method, 211-214
 - DeleteOrders method, 232-233
 - FullUpdateOrder method, 237-239
 - GetFullOrders methods, 195-196
 - GetFullOrders_By_Customer method, 205-207
 - GetOrderDetails methods, 193-195, 201-205
 - GetOrders methods, 184-189
 - implementing, 177-179
 - InsertOrderDetails method, 214-216
 - InsertOrders method, 234-237
 - sp_DeleteOrders stored procedure, 227-228
 - sp_InsertOrders stored procedure, 223-227
 - sp_UpdateOrders stored procedure, 220-223
 - testing update methods, 241
 - UpdateOrderDetails method, 216-218
 - UpdateOrders method, 228-232
 - using stored procedures in, 220-228
 - XML results, 190-193
 - XML Schema, 197-200
- page scope, 122
 - Parameters property, 32
 - passwords, using, 129

- pessimistic concurrency, 119
- PivotTable Service, 273
- pluggable architecture of XML, 132-133
- pooling, 125-127
- Prefix property, 74
- primitive data types, 301
- Private keyword, 310
- properties, 308
 - defining in class, 124-125
 - implementing, 312-315
- Property procedure, 312-313
- Protected keyword, 310
- Provider property, 23
- public fields, using in classes, 123-124
- Public keyword, 310-311

- Read() method, 53
- ReadXmlSchema() method, 89
- ReadXml() method, 88
 - using, 150-152
- RecordsAffected property, 39
- Recordset object, 10, 15
- recursion, 369, 384
- recursive triggers, 369
- RejectChanges() method, 89-90
- Relations property, 75
- relationships, maintaining, 12-14
- Reset() method, 90
- ROLLUP, 400
 - using, 401-404

- Schema Object Model, *see* SOM
- scope, 122-123
- Second Normal Form, 330-331
- security, 129-139
- SELECT, using, 376-381
- SelectCommand property, 60-61
- ServerVersion property, 23-24
- session scope, 122
- Set procedure, 314-315
- site, 121
- Site property, 75
- SOM, 140-141
- sp_executesql stored procedure, 392-395

- sp_Product_Employee_xtab stored procedure, 412-416
- sp_QuickSort stored procedure, 386-390
- SQL,
 - creating stored procedures with, 357-360
 - creating triggers with, 366-369
 - creating views with, 343-346
 - using, 376-383
- SQL Server .NET Data Provider, using to connect to database, 29-30
- SQL statements vs. stored procedures, 127
- SqlDataAdapter object, 54-57
 - methods, 62-66
 - properties, 57-61
- State property, 24
- stored procedure generator, 396-399
- stored procedures, 354
 - creating with SQL, 357-360
 - example of using, 361-363
 - executing, 361
 - managing, 356-357
 - reasons for using, 355-356
 - recursive, 384-390
 - vs. SQL statements, 127
- subclassing, 132
- system stored procedure, 354

- TableMappings property, 61
- Tables property, 75-76
- tables, relations between, 159-164
- Third Normal Form, 331-332
- ToString() method, 28, 37, 90-91
- Transaction property, 32
- triggers, 364
 - creating with SQL, 366-369
 - example of using, 370-372
 - managing, 365
 - nested, 370
 - reasons for using, 364-365
 - recursive, 369
- tuning, 128

- Update() method, 65-66

Index

- updateable view, 353-354
- UpdateCommand, 61
- variant data type, 246
- VB .NET, 299-300
 - implementing class in, 302-303
 - implementing inheritance in, 320-324
 - object orientation in, 301-324
- views, 336
 - creating with SQL, 343-346
 - example of using, 347-349, 349-353
 - managing, 338-339
 - reasons for using, 336-338
 - updateable, 353-354
 - using with Enterprise Manager, 340-342
- virtual table, *see* views
- Visual Basic .NET, *see* VB .NET
- Visual Basic, object-oriented support in, 300
- Visual Studio libraries, 288-289
- Web Forms, 107
 - data access in, 107-109
- Web Service, 175
 - designing, 175-176
 - functionality for Northwind example, 174-175
- Web Service namespace, 180-181
- WHERE clause, using with DELETE, 381-383
- Windows Forms applications, 99-100
 - data access in, 103-104
- Windows Services applications, 105
 - data access in, 105-106
- WriteXml() method, 91, 152-153
- WriteXmlSchema() method, 91-92
- XML, 131
 - extensibility of, 132
 - inferring in, 170-171
 - integrating with relational data, 143
 - integration with ADO .NET, 133-134
 - integration with .NET Framework, 131-134
 - performance of classes in, 133
 - pluggable architecture of, 132-133
 - using with DataSet, 144
 - using with MS SQL Server 2000, 143-144
 - writing, 152-154
- XML documents,
 - loading in DOM, 138-139
 - validating, 139-142
- XML Schema,
 - creating DataSet relational schema from, 164-170
 - data types, 141-142
- XML Web Service, 109-112
 - using DataSet in, 112-115
- XmlDataDocument,
 - synchronizing with DataSet, 157-159
 - using with DataSet, 156-157
- XPath, 132
- XSLT, 132

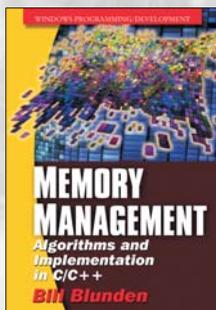
Looking for more?

Check out Wordware's market-leading Windows Programming/Development and Web Programming/Development Libraries featuring the following new releases.



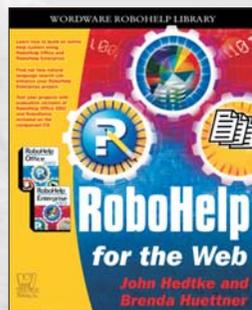
Search Engine Optimization with WebPosition Gold 2

1-55622-924-0 • \$49.95
7½ x 9¼ • 360 pp.



Memory Management Algorithms and Implementation in C/C++

1-55622-347-1 • \$59.95
6 x 9 • 392 pp.

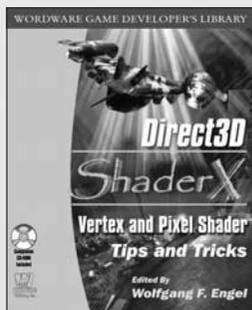


RoboHelp for the Web

1-55622-954-2 • \$49.95
7½ x 9¼ • 448 pp.

Look for these developer libraries and more from Wordware

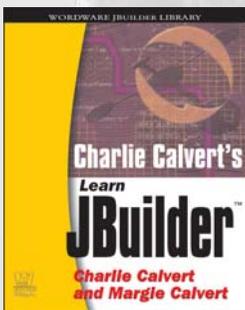
Game Developer's Library featuring:



Direct3D ShaderX Vertex and Pixel Shader Tips and Tricks

1-55622-041-3 • \$59.95
7½ x 9¼ • 520 pp.

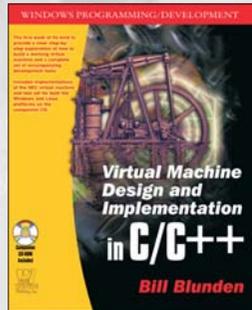
JBuilder Library featuring:



Charlie Calvert's Learn JBuilder

1-55622-330-7 • \$59.95
7½ x 9¼ • 912 pp.

Windows Programming featuring:



Virtual Machine Design and Implementation in C/C++

1-55622-903-8 • \$59.95
7½ x 9¼ • 688 pp.

Visit us online at www.wordware.com for more information. Use the following coupon code for online specials:

ADO-9658

TEAM LiNG - Live, Informative, Non-cost and Genuine!

About the CD

The CD-ROM included with this book contains examples that demonstrate ADO .NET topics discussed in the book. The examples are organized by chapter in the Code Samples folder.



Warning: By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement on the following page.

Opening the CD package makes this book nonreturnable.

CD/Source Code Usage License Agreement

Please read the following CD/Source Code usage license agreement before opening the CD and using the contents therein:

1. By opening the accompanying software package, you are indicating that you have read and agree to be bound by all terms and conditions of this CD/Source Code usage license agreement.
2. The compilation of code and utilities contained on the CD and in the book are copyrighted and protected by both U.S. copyright law and international copyright treaties, and is owned by Wordware Publishing, Inc. Individual source code, example programs, help files, freeware, shareware, utilities, and evaluation packages, including their copyrights, are owned by the respective authors.
3. No part of the enclosed CD or this book, including all source code, help files, shareware, freeware, utilities, example programs, or evaluation programs, may be made available on a public forum (such as a World Wide Web page, FTP site, bulletin board, or Internet news group) without the express written permission of Wordware Publishing, Inc. or the author of the respective source code, help files, shareware, freeware, utilities, example programs, or evaluation programs.
4. You may not decompile, reverse engineer, disassemble, create a derivative work, or otherwise use the enclosed programs, help files, freeware, shareware, utilities, or evaluation programs except as stated in this agreement.
5. The software, contained on the CD and/or as source code in this book, is sold without warranty of any kind. Wordware Publishing, Inc. and the authors specifically disclaim all other warranties, express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the disk, the program, source code, sample files, help files, freeware, shareware, utilities, and evaluation programs contained therein, and/or the techniques described in the book and implemented in the example programs. In no event shall Wordware Publishing, Inc., its dealers, its distributors, or the authors be liable or held responsible for any loss of profit or any other alleged or actual private or commercial damage, including but not limited to special, incidental, consequential, or other damages.
6. One (1) copy of the CD or any source code therein may be created for backup purposes. The CD and all accompanying source code, sample files, help files, freeware, shareware, utilities, and evaluation programs may be copied to your hard drive. With the exception of freeware and shareware programs, at no time can any part of the contents of this CD reside on more than one computer at one time. The contents of the CD can be copied to another computer, as long as the contents of the CD contained on the original computer are deleted.
7. You may not include any part of the CD contents, including all source code, example programs, shareware, freeware, help files, utilities, or evaluation programs in any compilation of source code, utilities, help files, example programs, freeware, shareware, or evaluation programs on any media, including but not limited to CD, disk, or Internet distribution, without the express written permission of Wordware Publishing, Inc. or the owner of the individual source code, utilities, help files, example programs, freeware, shareware, or evaluation programs.
8. You may use the source code, techniques, and example programs in your own commercial or private applications unless otherwise noted by additional usage agreements as found on the CD.



Warning: By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement.

Additionally, opening the CD package makes this book nonreturnable.