

**Robert Lair
Jason Lefebvre**

PURE ASP.NET

A Code-Intensive Premium Reference

- Covers ASP.NET and the Microsoft .NET Framework
- Comprehensive reference for ASP.NET in both VB.NET and C#
- Filled with practical, everyday examples
- Includes an in-depth ASP.NET Framework Reference



SAMS

PURE



ASP.NET

Robert Lair
Jason Lefebvre

SAMS

Copyright © 2002 by Sams

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32069-X

Library of Congress Catalog Card Number: 00-108321

Printed in the United States of America

First Printing: September 2001

04 03 02 01

4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

ASSOCIATE PUBLISHER

Jeff Koch

ACQUISITIONS EDITOR

Neil Rowe

DEVELOPMENT EDITOR

Mark Renfrow

MANAGING EDITOR

Matt Purcell

PROJECT EDITOR

Natalie Harris

COPY EDITOR

Kitty Jarrett

INDEXER

Tina Trettin

PROOFREADER

Jody Larsen

TECHNICAL EDITOR

John Timney

TEAM COORDINATOR

Vicki Harding

INTERIOR DESIGNER

Karen Ruggles

COVER DESIGNER

Aren Howell

PAGE LAYOUT

Susan Geiselman

Overview

Introduction

PART I OVERVIEW OF ASP.NET

- 1 ASP.NET and the Microsoft.NET Framework
- 2 The Common Language Runtime
- 3 ASP.NET Web Forms
- 4 ASP.NET Controls
- 5 List Controls
- 6 Validating User Input
- 7 Understanding Data Access with ADO.NET
- 8 Overview of Web Services

PART II DEVELOPING WITH ASP.NET

- 9 Building ASP.NET Pages with HTML and Web Controls
- 10 Encapsulating ASP.NET Page Functionality with User Controls
- 11 Working with ASP.NET List Controls
- 12 Working with ASP.NET Validation Controls
- 13 Data Access with ADO.NET
- 14 Building Components for ASP.NET
- 15 Building Web Services
- 16 Configuring and Optimizing an ASP.NET Application
- 17 Securing and Deploying an ASP.NET Application

PART III MICROSOFT .NET REFERENCE BY NAMESPACE

- 18 System.Collections Reference
 - 19 System.Data.SqlClient Reference
 - 20 System.Web Reference
 - 21 System.Web.UI.WebControls Reference
- Index

Contents

Introduction	1
--------------	---

PART I OVERVIEW OF ASP.NET 3

1 ASP.NET AND THE MICROSOFT .NET FRAMEWORK	5
Overview of Microsoft .NET	5
Goals of Microsoft .NET	5
Support for Previous Versions of ASP	5
Simpler Programming Model	6
Technologies of the Microsoft .NET Framework	6
Common Language Runtime Engine	6
ASP.NET Web Forms	7
ASP.NET Controls	7
ViewState Management	8
Validation Controls	8
Web Services	8
Benefits of the Microsoft .NET Framework	9
Fewer Lines of Code	9
Complete Compilation	10
Ease of Deployment	10
Web Settings and web.config	11
Output, Data, and Fragment Caching	11
2 THE COMMON LANGUAGE RUNTIME	13
Overview of the Common Language Runtime	13
Microsoft Intermediate Language	14
JIT Compilation	14
Assemblies	14
Manifests	14
Benefits of Assemblies	15
The Intermediate Language (IL) Disassembler	15
Garbage Collection in the	
Microsoft .NET Runtime	16
The Mark and Compact Algorithm	16
Generations	18
Manual Control of Garbage Collection	19
3 ASP.NET WEB FORMS	21
Server-Side Code Blocks	21
Compiling ASP.NET Web Forms	22

ASP.NET Web Form Events	26
Page Directives	26
Server-Side Includes	27
User Controls	28
Visual Studio and the Code Behind Method	28
4 ASP.NET CONTROLS	31
ASP.NET Controls Generate HTML	32
ViewState Management	34
Implementing ViewState Management	34
Turning Off ViewState Management to Increase Performance	35
HTML Controls	36
Web Controls	36
Benefits of Using Web Controls	36
The Web Control Base Class	37
Event Handling	37
Rich Controls	38
Custom Controls	38
4 LIST CONTROLS	39
Types of List Controls	39
Repeater	40
DataList	40
DataGrid	41
Benefits of Using List Controls	42
Simplified Development	42
Browser Independence	43
Built-in Support for Common Operations	43
How the List Controls Work	47
Data Binding to List Controls	47
6 VALIDATING USER INPUT	49
What Are Validation Controls?	49
Benefits of Using Validation Controls	50
Using Uplevel Versus Downlevel HTML Browsers	50
Eliminating Client-Side Scripting to	
Validate Input Data	50
Providing Consistent Error Reporting	51
Separating Control	51
Types of Validation Controls	51
The RequiredFieldValidator Control	51
The RangeValidator Control	51
The CompareValidator Control	52
The RegularExpressionValidator Control	52
The CustomValidator Control	52

Displaying Errors	52
Error Formatting	52
Validation Control Placement	53
7 UNDERSTANDING DATA ACCESS WITH ADO.NET	55
Benefits of ADO.NET	55
Interoperability	55
Scalability and Performance	56
The ADO.NET Object Model	56
The DataTable Class	57
The DataSet Class	57
The Connection Class	58
The Command Class	58
The DataReader Class	58
8 OVERVIEW OF WEB SERVICES	59
What Are Web Services?	59
Distributed Computing	59
SOAP for Communication	60
Why Use Web Services?	60
Standard Web Ports	60
Black Box Functionality	60
Site Functionality Exposure	60
Programming Model Flexibility	61
Class Library Support	61
Security	61
Improved End User Experience	61
Exposing Web Services	61
Discovering Web Services	62
Consuming Web Services	62
WSDL	63
Proxy Classes	63
Calling a Web Service	64
PART II DEVELOPING WITH ASP.NET 65	
9 BUILDING ASP.NET PAGES WITH HTML AND WEB CONTROLS	67
A Technical Overview of Web Controls	67
Core Web Controls	68
General Web Controls	68
Web Controls for Building Forms	83
10 ENCAPSULATING ASP.NET PAGE FUNCTIONALITY WITH USER CONTROLS	111
Creating a Simple User Control	111
Registering a User Control on a Page	112
Creating an Instance of a User Control	112

Adding Properties to a User Control	114
Adding Methods to a User Control	116
Fragment Caching	117
11 WORKING WITH ASP.NET LIST CONTROLS	119
Binding Data to List Controls	119
The DataSource Property	120
The DataBind() Method	120
Binding Expressions	121
Binding Examples	122
The Repeater List Control	135
Repeater Control Templates	135
Using Repeater Controls	135
The DataList List Control	139
DataList Control Templates	139
Using DataList Controls	140
Formatting a DataList Control with Styles	142
Creating a Columnar DataList Control	148
Selecting Items in a DataList Control	150
The DataGrid Control	154
The AutoGenerateColumns Property	154
Using the DataGrid Control	155
Formatting a DataGrid Control with Styles	157
Defining Columns in a DataGrid Control	161
Selecting Items in a DataGrid Control	166
Paging DataGrid Items	171
Sorting Items in a DataGrid Control	175
Creating a Master/Detail View	178
12 WORKING WITH ASP.NET VALIDATION CONTROLS	187
Properties and Methods Common to All Controls	187
Placing a Control on a Web Form	188
Formatting Error Messages	188
The RequiredField Validator	189
The Range Validator	193
The Compare Validator	195
The RegularExpression Validator	198
The Custom Validator	201
The ValidationSummary Web Control	205
13 DATA ACCESS WITH ADO.NET	211
Using the Connection Object	212
Using the Command Object	213
Using the DataReader Object	214
Instantiating the DataReader Object	215
Stepping Through a Result Set with Read()	218

Working with Stored Procedures	223
Retrieving a Dataset	223
Using Parameters	228
Working with Transactions	238
Beginning a Transaction	238
Rolling Back a Transaction	239
Rolling Back a Transaction to a Saved Point	240
Committing a Transaction	241
14 BUILDING COMPONENTS FOR ASP.NET	243
What Are Components?	243
Benefits of Using Components	243
Design Benefits	244
Implementation Benefits	244
Maintenance Benefits	244
Microsoft Windows DNA	244
Windows DNA and Microsoft .NET	245
Components and Microsoft .NET	245
Namespaces	245
Declaring Namespaces	246
Classes	247
Class Members	247
Static (Shared) and Instance Members	249
Member Accessibility	252
Compiling a Microsoft .NET Component	252
Accessing Components from ASP.NET Applications	253
COM Interoperability	255
Late-Bound COM References	256
Early-Bound COM References	256
COM+ Services: Using Transactions from Microsoft .NET	258
Committing a Transaction	259
Aborting a Transaction	259
Using AutoComplete	260
A Transaction Example	260
15 BUILDING WEB SERVICES	269
Creating a Web Service	269
Designing a Web Service Interface	269
Developing a Web Service	269
Building Web Services into	
an Existing Application	271
Exposing Useful Functionality	273
Consuming Web Services	274
Analyzing the WSDL Contract	275
Testing a Web Service	276
Generating a Proxy Class	278

Using Precompiled Web Services	282
Consuming Web Services from a Web Form	282
Consuming Web Services from a Windows Form	284
Generating a Discovery File	284
Dynamic Discovery	285
16 CONFIGURING AND OPTIMIZING AN ASP.NET APPLICATION	287
Configuring ASP.NET Applications	287
The machine.config File	287
The web.config File	288
Storing Application Settings	288
Custom Error Reporting	289
Optimizing ASP.NET Applications	291
The ASP.NET Caching Services	291
17 SECURING AND DEPLOYING AN ASP.NET APPLICATION	293
ASP.NET Installation Benefits	294
Side-by-Side DLL Execution	294
Deploying ASP.NET Pages	294
Creating the Virtual Directory	294
Deploying ASP.NET Applications by Using Standard Internet Protocols	295
Scripting Deployment	295
Deploying Components	295
Setting Up Staged Deployment	295
Three-Tier Hardware Requirements	296
Benefits of Staged Deployment	296
Securing an Application	296
Security by Design	296
Windows Authentication	297
PART III MICROSOFT .NET REFERENCE BY NAMESPACE 301	
18 SYSTEM.COLLECTIONS REFERENCE	303
The ArrayList Class	304
ArrayList.Capacity	306
ArrayList.Count	307
ArrayList.FixedSize	307
ArrayList.ReadOnly	308
ArrayList.Synchronized	309
ArrayList.Item	309
ArrayList.Adapter()	310
ArrayList.Add()	311
ArrayList.AddRange()	311
ArrayList.BinarySearch()	312
ArrayList.Clear()	313

ArrayList.Clone()	313
ArrayList.Contains()	314
ArrayList.CopyTo()	314
ArrayList.FixedSize()	315
ArrayList.GetRange()	316
ArrayList.IndexOf()	317
ArrayList.Insert()	317
ArrayList.InsertRange()	318
ArrayList.LastIndexOf()	319
ArrayList.ReadOnly()	320
ArrayList.Remove()	320
ArrayList.RemoveAt()	321
ArrayList.RemoveRange()	321
ArrayList.Repeat()	322
ArrayList.Reverse()	323
ArrayList.SetRange()	323
ArrayList.Sort()	324
ArrayList.Synchronized	325
ArrayList.ToArray	326
ArrayList.TrimToSize	326
The BitArray Class	326
BitArray.Count	327
BitArray.IsReadOnly	328
BitArray.IsSynchronized	328
BitArray.Item[]	329
BitArray.Length	330
BitArray.And()	330
BitArray.Clone()	331
BitArray.CopyTo()	332
BitArray.Get()	332
BitArray.Not()	333
BitArray.Or()	334
BitArray.Set()	335
BitArray.SetAll()	335
BitArray.Xor()	336
The Hashtable Class	336
Hashtable.Count	337
Hashtable.IsReadOnly	338
Hashtable.IsSynchronized	338
Hashtable.Item	339
Hashtable.Keys	340
Hashtable.Values	340
Hashtable.Add()	341
Hashtable.Clear()	341
Hashtable.Clone()	342

Hashtable.Contains()	343
Hashtable.ContainsKey()	343
Hashtable.ContainsValue()	344
Hashtable.CopyTo()	345
Hashtable.Remove()	346
Hashtable.Synchronized()	346
The Queue Class	347
Queue.Count	348
Queue.IsReadOnly	348
Queue.IsSynchronized	349
Queue.Clear()	349
Queue.Clone()	350
Queue.Contains()	351
Queue.CopyTo()	351
Queue.Dequeue()	352
Queue.Enqueue()	353
Queue.Peek()	353
Queue.Synchronized()	354
Queue.ToArray()	354
The SortedList Class	355
SortedList.Capacity	356
SortedList.Count	357
SortedList.IsReadOnly	357
SortedList.IsSynchronized	358
SortedList.Item	359
SortedList.Keys	359
SortedList.Values	360
SortedList.Add()	361
SortedList.Clear()	361
SortedList.Clone()	362
SortedList.Contains()	362
SortedList.ContainsKey()	363
SortedList.ContainsValue()	364
SortedList.CopyTo()	365
SortedList.GetByIndex()	365
SortedList.GetKey()	366
SortedList.GetKeyList()	367
SortedList.GetValueList()	367
SortedList.IndexOfKey()	368
SortedList.IndexOfValue()	369
SortedList.Remove()	369
SortedList.RemoveAt()	370
SortedList.SetByIndex()	371
SortedList.Synchronized()	371
SortedList.TrimToSize()	372

The Stack Class	373
Stack.Count	373
Stack.IsReadOnly	374
Stack.IsSynchronized	375
Stack.Clear()	375
Stack.Clone()	376
Stack.Contains()	376
Stack.CopyTo()	377
Stack.Peek()	378
Stack.Pop()	378
Stack.Push()	379
Stack.Synchronized()	380
Stack.ToArray()	380
19 SYSTEM.DATA.SQLCLIENT REFERENCE	383
The SqlCommand Class	385
SqlCommand.CommandText	385
SqlCommand.CommandTimeout	386
SqlCommand.CommandType	386
SqlCommand.Connection	387
SqlCommand.Parameters	388
SqlCommand.Transaction	389
SqlCommand.Cancel()	390
SqlCommand.CreateParameter()	390
SqlCommand.ExecuteNonQuery()	391
SqlCommand.ExecuteReader()	392
SqlCommand.ExecuteScalar()	392
SqlCommand.ExecuteXmlReader()	393
SqlCommand.ResetCommandTimeout()	394
The SqlConnection Class	395
SqlConnection.ConnectionString	395
SqlConnection.ConnectionTimeout	396
SqlConnection.Database	396
SqlConnection.DataSource	397
SqlConnection.PacketSize	397
SqlConnection.ServerVersion	398
SqlConnection.State	398
SqlConnection.WorkstationId	399
SqlConnection.BeginTransaction()	399
SqlConnection.ChangeDatabase()	400
SqlConnection.Close()	401
SqlConnection.CreateCommand()	401
SqlConnection.Open()	402
The SqlDataReader Class	402
SqlDataReader.FieldCount	404

SqlDataReader.IsClosed	405
SqlDataReader.Item	406
SqlDataReader.RecordsAffected	407
SqlDataReader.Close()	408
SqlDataReader.GetBoolean()	409
SqlDataReader.GetByte()	410
SqlDataReader.GetBytes()	411
SqlDataReader.GetChar()	412
SqlDataReader.GetChars()	413
SqlDataReader.GetDataTypeName()	414
SqlDataReader.GetDateTime()	415
SqlDataReader.GetDecimal()	416
SqlDataReader.GetDouble()	417
SqlDataReader.GetFieldType()	418
SqlDataReader.GetFloat()	419
SqlDataReader.GetGuid()	420
SqlDataReader.GetInt16()	421
SqlDataReader.GetInt32()	423
SqlDataReader.GetInt64()	424
SqlDataReader.GetName()	425
SqlDataReader.GetOrdinal()	426
SqlDataReader.GetSchemaTable()	427
SqlDataReader.GetSqlBinary()	428
SqlDataReader.GetSqlBit()	429
SqlDataReader.GetSqlByte()	430
SqlDataReader.GetSqlDateTime()	431
SqlDataReader.GetSqlDecimal()	432
SqlDataReader.GetSqlDouble()	433
SqlDataReader.GetSqlGuid()	434
SqlDataReader.GetSqlInt16()	435
SqlDataReader.GetSqlInt32()	436
SqlDataReader.GetSqlInt64()	437
SqlDataReader.GetSqlMoney()	438
SqlDataReader.GetSqlSingle()	439
SqlDataReader.GetSqlString()	440
SqlDataReader.GetSqlValue()	441
SqlDataReader.GetString()	442
SqlDataReader.GetValue()	443
SqlDataReader.IsDBNull()	444
SqlDataReader.NextResult()	446
SqlDataReader.Read()	447
The SqlDbType Class	448
The SqlParameter Class	448
SqlParameter.DbType	449
SqlParameter.Direction	449

SqlParameter.IsNull 450
SqlParameter.ParameterName 450
SqlParameter.Precision 451
SqlParameter.Scale 451
SqlParameter.Size 452
SqlParameter.Value 452
The SqlParameterCollection Class 453
SqlParameterCollection.Count 453
SqlParameterCollection.Item 455
SqlParameterCollection.Add() 456
SqlParameterCollection.Clear() 457
SqlParameterCollection.Contains() 458
SqlParameterCollection.IndexOf() 459
SqlParameterCollection.Insert() 461
SqlParameterCollection.Remove() 462
SqlParameterCollection.RemoveAt() 463
The SqlTransaction Class 465
SqlTransaction.IsolationLevel 465
SqlTransaction.Commit() 466
SqlTransaction.Rollback() 466
SqlTransaction.Save() 467
20 SYSTEM.WEB REFERENCE 469
The HttpBrowserCapabilities Class 469
HttpBrowserCapabilities.ActiveXControls 470
HttpBrowserCapabilities.AOL 471
HttpBrowserCapabilities.BackgroundSounds 471
HttpBrowserCapabilities.Beta 472
HttpBrowserCapabilities.Browser 472
HttpBrowserCapabilities.CDF 472
HttpBrowserCapabilities.Cookies 473
HttpBrowserCapabilities.Crawler 473
HttpBrowserCapabilities.EcmaScriptVersion 474
HttpBrowserCapabilities.Frames 474
HttpBrowserCapabilities.JavaApplets 475
HttpBrowserCapabilities.JavaScript 475
HttpBrowserCapabilities.MajorVersion 476
HttpBrowserCapabilities.MinorVersion 476
HttpBrowserCapabilities.MsDomVersion 477
HttpBrowserCapabilities.Platform 477
HttpBrowserCapabilities.Tables 478
HttpBrowserCapabilities.Type 478
HttpBrowserCapabilities.VBScript 479
HttpBrowserCapabilities.Version 479
HttpBrowserCapabilities.W3CDomVersion 480

HttpBrowserCapabilities.Win16	480
HttpBrowserCapabilities.Win32	481
The HttpCookie Class	481
HttpCookie.Domain	484
HttpCookie.Expires	484
HttpCookie.HasKeys	485
HttpCookie.Name	485
HttpCookie.Path	485
HttpCookie.Secure	486
HttpCookie.Value	486
HttpCookie.Values	487
The HttpRequest Class	487
HttpRequest.AcceptTypes	489
HttpRequest.ApplicationPath	490
HttpRequest.Browser	490
HttpRequest.ClientCertificate	490
HttpRequest.ContentEncoding	491
HttpRequest.ContentLength	491
HttpRequest.ContentType	492
HttpRequest.Cookies	492
HttpRequest.FilePath	493
HttpRequest.Files	493
HttpRequest.Form	496
HttpRequest.Headers	497
HttpRequest.HttpMethod	497
HttpRequest.IsAuthenticated	498
HttpRequest.IsSecureConnection	498
HttpRequest.Params	499
HttpRequest.Path	499
HttpRequest.PathInfo	500
HttpRequest.PhysicalApplicationPath	500
HttpRequest.PhysicalPath	501
HttpRequest.QueryString	501
HttpRequest.RawUrl	501
HttpRequest.RequestType	502
HttpRequest.ServerVariables	502
HttpRequest.TotalBytes	503
HttpRequest.Url	503
HttpRequest.UrlReferrer	504
HttpRequest.UserAgent	504
HttpRequest.UserHostAddress	505
HttpRequest.UserHostName	505
HttpRequest.UserLanguages	506
HttpRequest.MapPath()	507
HttpRequest.SaveAs()	507

The HttpResponse Class	508
HttpResponse.Buffer	509
HttpResponse.BufferOutput	509
HttpResponse.CacheControl	510
HttpResponse.Charset	510
HttpResponse.ContentEncoding	511
HttpResponse.ContentType	511
HttpResponse.Cookies	511
HttpResponse.Expires	512
HttpResponse.ExpiresAbsolute	512
HttpResponse.IsClientConnected	513
HttpResponse.Status	513
HttpResponse.StatusCode	514
HttpResponse.StatusDescription	514
HttpResponse.SuppressContent	515
HttpResponse.AddHeader()	515
HttpResponse.AppendCookie()	516
HttpResponse.AppendHeader()	516
HttpResponse.AppendToLog()	517
HttpResponse.BinaryWrite()	517
HttpResponse.Clear()	518
HttpResponse.ClearContent()	518
HttpResponse.ClearHeaders()	519
HttpResponse.Close()	519
HttpResponse.End()	519
HttpResponse.Flush()	520
HttpResponse.Pics()	520
HttpResponse.Redirect()	521
HttpResponse.Write()	521
HttpResponse.WriteFile()	522
21 SYSTEM.WEB.UI.WEBCONTROLS REFERENCE	523
The WebControl Class	523
WebControl.AccessKey	524
WebControl.Attributes	524
WebControl.BackColor	525
WebControl.BorderColor	525
WebControl.BorderStyle	526
WebControl.BorderWidth	526
WebControl.ClientID	527
WebControl.CssClass	527
WebControl.Enabled	528
WebControl.EnableViewState	528
WebControl.Font	528
WebControl.ForeColor	529

WebControl.Height	529
WebControl.ID	530
WebControl.Page	530
WebControl.Parent	531
WebControl.TabIndex	531
WebControl.ToolTip	532
WebControl.Visible	532
WebControl.Width	532
The HyperLink Class	533
HyperLink.ImageUrl	533
HyperLink.NavigateUrl	534
HyperLink.Target	534
HyperLink.Text	535
The Button Class	535
Button.CommandArgument	535
Button.CommandName	536
Button.Text	536
The Calendar Class	537
Calendar.CellPadding	538
Calendar.CellSpacing	539
Calendar.DayHeaderStyle	539
Calendar.DayNameFormat	540
Calendar.DayStyle	540
Calendar.FirstDayOfWeek	541
Calendar.NextMonthText	541
Calendar.NextPrevFormat	542
Calendar.NextPrevStyle	542
Calendar.OtherMonthDayStyle	542
Calendar.PrevMonthText	543
Calendar.SelectedDate	543
Calendar.SelectedDayStyle	544
Calendar.SelectionMode	544
Calendar.SelectMonthText	545
Calendar.SelectorStyle	545
Calendar.SelectWeekText	546
Calendar.ShowDayHeader	546
Calendar.ShowGridLines	547
Calendar.ShowNextPrevMonth	547
Calendar.ShowTitle	548
Calendar.TitleFormat	548
Calendar.TitleStyle	548
Calendar.TodayDayStyle	549
Calendar.TodaysDate	549
Calendar.VisibleDate	550
Calendar.WeekendDayStyle	550

The Label Class	551
Label.Text	551
The Image Class	551
Image.AlternateText	552
Image.ImageAlign	552
Image.ImageUrl	553
The Panel Class	553
Panel.BackImageUrl	553
Panel.HorizontalAlign	554
Panel.Wrap	554
The TextBox Class	555
TextBox.AutoPostBack	555
TextBox.Columns	556
TextBox.MaxLength	556
TextBox.ReadOnly	557
TextBox.Rows	557
TextBox.Text	558
TextBox.TextMode	558
TextBox.Wrap	558
The CheckBox Class	559
CheckBox.AutoPostBack	559
CheckBox.Checked	560
CheckBox.Text	560
CheckBox.TextAlign	561
The ImageButton Class	561
ImageButton.AlternateText	562
ImageButton.CommandArgument	562
ImageButton.CommandName	563
ImageButton.ImageAlign	563
ImageButton.ImageUrl	564
The LinkButton Class	564
LinkButton.CommandArgument	564
LinkButton.CommandName	565
LinkButton.Text	566
The RadioButton Class	566
RadioButton.AutoPostBack	566
RadioButton.Checked	567
RadioButton.GroupName	567
RadioButton.Text	568
RadioButton.TextAlign	568
The BaseValidator Class	569
BaseValidator.ControlToValidate	569
BaseValidator.Display	570
BaseValidator.EnableClientScript	570
BaseValidator.ErrorMessage	571

BaseValidator.IsValid	571
BaseValidator.Text	572
The CompareValidator Class	572
CompareValidator.ControlToCompare	572
CompareValidator.Operator	573
CompareValidator.ValueToCompare	573
The CustomValidator Class	574
CustomValidator.ClientValidationFunction	574
The RangeValidator Class	575
RangeValidator.MaximumValue	575
RangeValidator.MinimumValue	575
The RegularExpressionValidator Class	576
RegularExpressionValidator.ValidationExpression	576
The RequiredFieldValidator Class	577
RequiredFieldValidator.InitialValue	577
The ValidationSummary Class	577
ValidationSummary.DisplayMode	578
ValidationSummary.HeaderText	578
ValidationSummary.ShowMessageBox	579
ValidationSummary.ShowSummary	579
Selected Static Classes	580
The FontInfo Class	580
The ImageAlign Class	580
The TextAlign Class	581
The BorderStyle Class	581
The ValidationSummaryDisplayMode Class	581
Index	583

About the Author

Robert Lair is a nationally known developer, technical writer, and trainer. He studied computer science and engineering at Wright State University in Dayton, Ohio, and has been working in the area of software development for more than five years. His latest achievements in the software industry include his work on IBuySpy (Vertigo Software), the premiere demo application that shows the new features of ASP.NET. Other applications Robert has worked on in his career include ProposalMaster and RFPMaster (The Sant Corporation), OpenTable.com (Vertigo Software), and Shoes.NET (Vertigo Software), the premiere demo application shown at DevDays 2000 to show off the Microsoft .NET enterprise servers. He has worked with a number of major clients, including Microsoft, MCI, Williams Communications, and Prognostics.

Robert has also provided training at technical conferences, discussing a number of topics on Microsoft .NET and ASP.NET, as well as how to increase the efficiency of the software development process by simplifying the maintenance of complex applications. Conferences at which he has spoken include ASP Connections and the ASP Developer's Conference. He has also presented a number of internal technical learning sessions while working at Vertigo Software and the Sant Corporation.

Robert has written numerous articles for nationally known magazines such as *Visual C++ Developer's Journal*, *Visual Basic Programmer's Journal*, and *.NET Developer*. His articles have covered a wide variety of topics, including C#, Microsoft .NET, ASP.NET, and Visual Basic.NET. Robert is also the author of a monthly column in *.NET Developer* that takes an in-depth look at the IBuySpy application.

Jason Lefebvre is currently working as a developer at the direct marketing services company Protocol (www.protocolusa.com). He studied mathematics and computer science at Wright State University in Dayton, Ohio. He has been working in the area of software Development for more than five years. His latest achievements in the software industry include his work on IBuySpy (Vertigo Software), the premiere demo application to show the new features of ASP.NET. Other applications Jason has worked on in his career include ProposalMaster and RFPMaster (The Sant Corporation), OpenTable.com (Vertigo Software), and Shoes.NET (Vertigo Software), the premiere demo application shown at DevDays 2000 to show off the Microsoft .NET enterprise servers. He has worked with a number of major clients, including Microsoft, MCI, and Williams Communications.

Jason has written numerous articles for nationally known magazines such as *Visual C++ Developer's Journal* on a wide variety of topics, including C#, Microsoft .NET, ASP.NET, and Visual Basic.NET.

About the Technical Editor

John Timney lives with his wife, Philippa, in the small town of Chester-Le-Street in the North of England. He is a postgraduate student at Nottingham University, having earned a master's degree in information technology following a bachelor's degree in management systems, and a postgraduate diploma in information technology from Humberside University. John specializes in Internet solutions, and his computing expertise has gained him a Microsoft MVP (Most Valuable Professional) award. His hobbies include the martial arts, and he has black belts in two different styles of karate.

Dedication

This book is dedicated to my loving wife, Debi, who gives me encouragement, inspiration, and drive, and who sacrificed our time together while I worked on this book. I would also like to dedicate it to my first child, Max, who is on the way and whom I cannot wait to meet and hold for the first time! I love you both very much!

—Robert Lair

In loving memory of my grandfather, Raymond Lefebvre. Your wisdom, kindness, and insights are dearly missed.

—Jason Lefebvre

Acknowledgments

We would like to thank the entire Sams team for all the help and guidance they provided us on this project. Specifically, we'd like to thank our superb acquisitions editor, Neil Rowe, who kept us focused throughout the project. A special thanks also goes out to our development editor, Mark Renfrow, who helped to ensure that this book was the best it could be. We'd also like to thank everyone else at Sams who worked on the book, including Natalie Harris, Kitty Jarrett, John Timney, and Meggo Barthlow.

While writing this book, several other people have helped us in numerous ways. A special thanks to Minh Truong, for the many late-night MSN Messenger conversations to discuss the latest and greatest technologies covered in this book. We would also like to thank Justin Rogers for his help on some of the technical issues we faced. In addition, we would like to thank Mike Amundsen for his book-writing advice. Also, major thanks and kudos to Susan Warren, Scott Guthrie, and the rest of the ASP.NET team for their outstanding work and dedication.

Additionally, we'd like to thank the folks at Blizzard Entertainment (www.blizzard.com), for all their polished, well-crafted, and (above all else) fun games. Bob would also like to thank Pepsi Co. for Mountain Dew, which helped him write long into the night (and sometimes morning).

Last, but by no means least, we would like to thank you for purchasing this book. We hope you find this book very helpful and enjoyable.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an associate publisher for Sams, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax: 317-581-4770

Email: feedback@sampsublishing.com

Mail:
Jeff Koch
Sams Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

Introduction

Before you begin reading Pure ASP.NET, you should understand how the book is organized. This will help you determine what you want to read and when you want to read it. This introduction provides a brief overview of the book's organization.

This book is written for developers with a fundamental understanding of programming structures and at least a casual understanding of HTML. However, this book assumes no prior knowledge of any part of the Microsoft .NET framework.

This book is divided into three logical parts. Part I, “Overview of ASP.NET,” includes eight chapters meant for developers that are new to ASP.NET and need a high-level overview of the technologies it encompasses. There is very little code in Part I; therefore, almost any developer should be able to understand the basic concepts of ASP.NET after reading this section.

Part II, “Developing with ASP.NET,” is the core of the book. These chapters build on the material covered in Part I and implement the topics discussed. In this part, you will work with Web forms and Web services built from scratch. You will also learn the proper techniques for building components as well as user controls. Part II is code intensive, yet it also includes discussion of the techniques needed.

Part III, “Microsoft .NET Reference by Namespace,” provides a quick reference for the System.Collections, System.Data.SqlClient, System.Web, and System.Web.UI.WebControls namespaces. This is the most code-intensive section of the book and contains very little discussion of concepts. It contains examples that show you exactly how to use the properties and methods found in some of the most important Microsoft .NET namespaces.

PART I

OVERVIEW OF ASP.NET

- 1 ASP.NET and the Microsoft.NET Framework
- 2 The Common Language Runtime
- 3 ASP.NET Web Forms
- 4 ASP.NET Controls
- 5 List Controls
- 6 Validating User Input
- 7 Understanding Data Access with ADO.NET
- 8 Overview of Web Services



CHAPTER 1

ASP.NET and the Microsoft .NET Framework

Overview of Microsoft .NET

With the introduction of the Microsoft.NET framework, Microsoft has committed itself to maintaining its position on the cutting edge of Internet development. The new framework addresses the current paradigm shift away from standard Web applications and toward a truly integrated user experience on the Internet by providing a number of new tools.

To ensure wide acceptance, the Microsoft.NET platform embraces open standards such as eXtensible Markup Language(XML) and Simple Object Access Protocol(SOAP). This enables ASP.NET to easily interface with applications on other platforms that support these open standards.

Goals of Microsoft .NET

The two main goals of Microsoft.NET are to provide support for previous versions of Active Server Pages (ASP) and to provide a simpler programming model.

Support for Previous Versions of ASP

One goal of the Microsoft.NET platform is ease of integration between new ASP.NET pages and pre ASP.NET pages, such as ASP 3.0. The two easily coexist on the same machine. In fact, the ASP.NET processor only attempts to compile and run files types that have been specified during the Microsoft .NET framework installation, such as .aspx and .asmx files. Thus, you can use both ASP and ASP.NET pages within the same application.

In addition, an ASP page can be converted to an ASP.NET page easily, usually with only minor changes to the code. Further easing the transition from ASP to ASP.NET is the fact that you can instantiate and use Component Object Model (COM) and COM+ objects on ASP.NET pages by using the interoperability features of the Microsoft .NET framework. For more information on working with COM objects, see Chapter 14, “Building Components for ASP.NET.”

Simpler Programming Model

Microsoft.NET provides a simpler programming model than COM and COM+. Some of the most confusing and cumbersome aspects of COM, such as the system registry and globally unique identifiers(GUID), are still accessible in, but are not an active part of, the Microsoft.NET programming model. Developers no longer need to worry about registering their components in the registry, and more importantly, they do not have to be concerned with keeping binary compatibility between different versions of their components.

Technologies of the Microsoft .NET Framework

The Microsoft .NET framework makes several new technologies available to ASP.NET. The Common Language Runtime engine gives ASP.NET true platform and language independence. It also offers a legion of server-side controls that enable you to create applications much more quickly.

Common Language Runtime Engine

One of the most significant pieces of the Microsoft.NET platform is the creation of a Common Language Runtime engine. In the Microsoft.NET platform, code is compiled not to assembly machine code but to an intermediate code called Microsoft Intermediate Language (MSIL). This allows several benefits that will be central to the Microsoft.NET platform’s success. This is covered in more detail in Chapter 2, “The Common Language Runtime.”

The Common Language Runtime engine provides two main benefits: platform independence and language independence.

Platform Independence

Because ASP.NET source code is not compiled to binaries for a particular platform, it is platform independent. In the future you might see common language runtime engines for other platforms as well, such as Linux or Macintosh systems. With ASP.NET, application code does not need to be recompiled between platforms. For example, you could develop on Windows 2000, taking advantage of Visual Studio.NET and other rich development tools, and then compile and deploy directly to a Unix-based operating system in order to satisfy customer requirements.

Language Independence

It does not matter which language you choose to code with in Microsoft.NET. Because all languages compile to the same intermediate code, the choice of which to use is a personal preference. In fact, different parts of the same application can be created from different languages. A class written in C# can be referenced from a block of Visual Basic.NET code just as easily as if the class had been written in Visual Basic.NET.

At the time this book was written, only four language choices were available for Microsoft.NET development: C#, Visual Basic.NET, JScript, and Managed C++. However, a number of efforts are under way to create compilers for a number of different languages, including Perl and a variety of others.

ASP.NET Web Forms

Central to the Microsoft.NET platform are *Web forms*, also referred to as *ASP.NET pages*. A Web form is basically a container item for ASP.NET controls and other Web form elements such as server-side script, as well as standard HTML and other client-side code.

ASP.NET Controls

ASP.NET offers a number of rich server-side controls that will help developers create Web applications that can be used in a number of browsers. By automatically detecting the type of browser that is making the request and generating correct HTML for that browser, ASP.NET controls remove the hassle of coding for multiple Web browsers, which in the past required a significant amount of coding time.

ASP.NET controls can generate large amounts of HTML to the client using only a single line of code on the server. This speeds up development and increases the readability of the code. Also, because the existing ASP.NET controls have been tested already, debugging an application is easy.

Developers are not limited to using the existing Microsoft controls. It is simple to create your own custom server controls, using a language such as C# or Visual Basic.NET. You create a control by determining the type of browser requesting the control and outputting HTML that is customized for that particular browser. Creating your own controls can be time-consuming. However, having any browser-specific display logic encapsulated in a control rather than strewn across your application is superior for several reasons:

- As new versions of browsers appear, you only need to update each of your custom controls. After the code in your control is updated and tested, you can be sure that your application will appear correctly in the updated browser.
- You can use your custom control across applications without having to be concerned about any browser-specific display logic in the Web form. It's all handled by your control.

ViewState Management

One of the greatest difficulties many developers face in designing Web applications is maintaining state (that is, making sure that values persist across multiple forms even when reloading the same page). Standard ASP marginally improved this over compiled Common Gateway Interface (CGI) applications (written in languages such as C and C++) by providing easy access to cookies, Session objects. However, because there could be serious performance issues involved with these methods, many developers maintained state by using hidden form elements.

In ASP.NET applications, maintaining state is no longer an arduous task, thanks to ViewState management. ViewState management is a built-in ability of Web Forms to automatically remember the values of the controls contained on a form. This eliminates the need to manually maintain the state of form elements.

Validation Controls

When developing a Web application, a great deal of time must be spent in creating the client-side script necessary to validate user input. Microsoft.NET provides a number of built-in validation controls that allow developers to easily validate user input with no client-side code at all. Using these validation controls, developers can mark form elements as required and can set length, comparison, or type requirements. Microsoft.NET even includes a Perl-based regular expression language that can be used to validate more advanced field elements (for example, making sure that a phone number field contains only numbers with dashes). If you have more advanced validation requirements than the built-in validation controls provide, you can create your own custom validation control with just a few lines of code.

Web Services

Until now, it has been fairly difficult to call remote methods on one Web server from another Web server. However, by using Web services, developers can create components whose methods can be accessed by applications that reside on another Web server. Figure 1.2 shows how a web service can be exposed and consumed by many heterogeneous clients. Because this is done using standard protocols such as XML and SOAP, Web services make it much easier for sites to communicate and share data with one another than ever before.

In addition, Web services offer many opportunities for code reuse and operate over standard Internet protocols.

Code Reuse

Developers can create public methods that are often reused between pages and applications by creating a Web Service that can be used by all who need it.

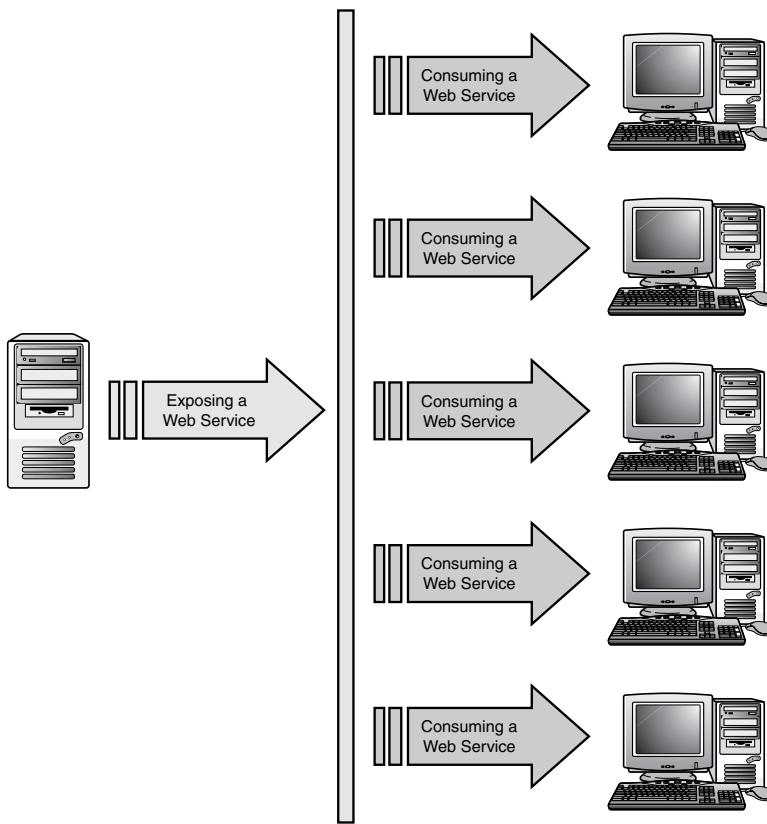


Figure 1.1

Exposing and consuming Web services.

Support for Standard Internet Protocols

To ensure that Web Services will be available to a wide variety of clients, Microsoft.NET embraces standard Internet protocols. The XML Simple Object Activation Protocol (SOAP) provides the standard format for passing data that enables Web Services to be used by remote clients running on different operating systems.

Benefits of the Microsoft .NET Framework

The Microsoft.NET framework provides a number of benefits. This section gives an overview of some of these benefits in detail.

Fewer Lines of Code

Through the use of Web controls, developers will spend less time coding minor user interface points; for example, developers will spend less time coding minor user interface points, such as extensive work to make a page function well in any browser that is HTML 3.2 compliant. This gives the developer more time to concentrate on application design and the general flow of the application.

In addition, due to the strongly typed nature of the Microsoft .NET framework languages, ASP.NET forces developers to use more sound development techniques.

Many ASP application problems derive from the loosely typed languages used. For instance, if you need to assign the value of an integer variable to a string variable in ASP, you simply use the = operator, and the integer is converted to a string behind the scenes. In ASP.NET, this is absolutely not allowed. You must convert the integer to type `string` before assigning it to the string variable. This sounds difficult, but is as simple as using the `ToString()` method, which is common to all integers in the Microsoft .NET framework:

```
string myVar = 4.ToString();
```

Strongly typed variables help to eliminate problems with type conversions that occur at runtime. The more application problems that can be solved at compile time instead of runtime, the more robust your application will be.

Complete Compilation

Another significant part of the Microsoft.NET framework is that everything is compiled, including ASP.NET pages themselves. When a page is requested for the first time since any modification, any server-side tags (such as Web controls), Web forms, and server-side blocks of code are compiled, and then the response is created and sent to the client. Subsequent hits on the same page from any user are much quicker because the page is already compiled.

Ease of Deployment

One of the most intriguing benefits to the Microsoft.NET framework is the deployment model. In traditional ASP, to deploy a Web application, several actions must be performed. The ASP pages must be uploaded, and any components that are part of the application must be uploaded and registered with the operating system. This can be accomplished through the use of elaborate Windows scripting host (.wsh) files that script all these actions. These scripts take time to create and are typically used only for large applications. For smaller applications, especially in an environment that is hosting a number of sites, application files are usually uploaded and then components are sent to the administrator for registration.

In designing the Microsoft .NET framework, Microsoft considered these problems with ASP and came up with an alternative approach. With Microsoft .NET, components can be compiled remotely on your machine and then uploaded with the pages and placed in the \bin directory in the root of the application. No registration is required, unless you need to register a legacy COM object. The upload can be performed over a standard File Transfer Protocol (FTP) connection to the server. No local server administration is required.

Also, deploying the Web application doesn't require that any services be stopped. The ASP.NET processor does not lock any files. The compiled component can be copied directly on top of the currently running component file without a problem. The

ASP.NET processor simply uses the version of the component or page in memory to finish serving current requests, and it reloads the newer version of the component or page into memory for new requests.

Web Settings and web.config

Configuration for Microsoft.NET applications is done via `web.config`, an XML-based text file that is stored in the same directory as the rest of the Web application files. The ASP.NET configuration methodology has many benefits that help make installing and configuring ASP.NET applications efficient and quick. The `web.config` file is an XML document. This makes it human readable, as well as programmatically modifiable. Also, because the `web.config` file can be modified by using any text editor, site configuration changes can be performed quickly, without any elaborate scripting or direct access to the server.

The Hierarchical Configuration Structure

A `web.config` file can be stored in the same directory as the rest of the Web files and control the configuration settings for files in its directory as well as all subdirectories. Because you can have multiple `web.config` files in an application, you can configure different parts of the application uniquely.

Human Readability

Because `web.config` is an XML-based text file, it is human readable. This makes modifying the configuration settings very simple, both for developers creating the system and for administrators maintaining the system. Modifications can be made with a simple text editor or XML editor. The ASP.NET configuration system makes deployment of a Microsoft.NET application simple. System administrators can use `xcopy` or `ftp` to copy `web.config`, along with any Web forms or components, to the Web server.

Automatic Detection of Changes

When the `Config.web` file is modified, the system detects the changes immediately. There is no need to restart the Web application.

Output, Data, and Fragment Caching

One of the most time-consuming and costly tasks in a Web application is database access. When your application experiences a great deal of user traffic, network latency, and the ability of the database server to process requests for data, it can quickly make database access become a bottleneck in your application. The Microsoft.NET framework provides a solution to this problem in the form of caching. *Caching* is the process of putting commonly accessed data into memory for quicker subsequent retrieval. The Microsoft.NET framework supports three types of caching: output caching, data caching, and fragment caching.

Output Caching

Output caching, also referred to as *page caching*, provides a way to place a page's output into the ASP.NET cache for a specified duration. If a user requests the cached page

during the specified period, the cached version is sent to the user. This provides an incredible performance increase for pages that contain infrequently changed data.

Data Caching

Data caching provides a way for you to place responses from specific database queries into the ASP.NET cache. Subsequent identical queries are served directly from the in-memory cache. This much explicit control over the caching of data gives you much more power to enhance the performance of applications. The data cache can expire at predetermined intervals or automatically whenever the data in the database changes.

Output and data caching are extremely easy to use and implement, and they can yield a huge performance increase for any ASP.NET application, especially as server traffic increases.

Fragment Caching

Fragment caching is related to output caching except that instead of caching the entire page, you have the ability to cache just a portion of the page. This is meant to allow caching on pages that have some dynamic data. A good example of fragment caching is pages that have built-in personalization. Because the pages are personalized, they will change constantly, depending on the user. Therefore, output caching would not provide a solution. Fragment caching allows you to cache the portions of the page that are not personalized, which will help to improve the performance of the Web application.

CHAPTER 2

The Common Language Runtime

Overview of the Common Language Runtime

All compiled programming languages require a runtime that provides the current architecture with the details on how to execute its code. Until Microsoft .NET, the problem has been that every programming language uses its own runtime. For example, if you want to develop with Visual Basic, you must make sure that the system that will be running the application has the Visual Basic runtime installed. However, having the Visual Basic runtime does not mean that the system can access an application written in Java. To run Java applications, the system would also require a Java virtual machine.

Microsoft .NET solves this problem by providing a common runtime that all the Microsoft .NET languages can use. Any system with the Microsoft .NET Common Language Runtime can run any language that is Microsoft .NET compatible.

Another benefit this provides is the ability to interoperate between different languages. Because all the languages share the same runtime, they all work well together. Therefore, you can implement an object that was written in C# from Visual Basic.NET. The same goes for any other Microsoft .NET language.

NOTE

Because ASP.NET pages are compiled, you are not permitted to mix languages on a given .aspx page. This is a change from classic ASP, which does allow the mixing of

languages. You can get around this by using user controls, which are discussed in Chapter 10, “Encapsulating ASP.NET Pages with Pagelets.”

Microsoft Intermediate Language

When you compile a Microsoft .NET language, the compiler generates code written in the Microsoft Intermediate Language (MSIL). MSIL is a set of instructions that can quickly and efficiently be translated into native code.

JIT Compilation

In order for a Microsoft .NET application to run, the MSIL code must be translated into native machine code. Instead of compiling the application at development time, the intermediate language is compiled “just in time” (JIT) into native code when the application or component is run. Two JIT compilers are provided with the Microsoft .NET runtime: the standard JIT compiler and the EconoJIT compiler. The EconoJIT compiler compiles faster than the standard JIT compiler, but the code it produces is not as optimized as that from the standard JIT compiler.

NOTE

In most circumstances, the standard JIT compiler should be used. However, because the standard JIT compiler requires quite a lot of RAM, for platforms that have less RAM, you might want to use the EconoJIT compiler, which requires fewer system resources than the JIT compiler.

Unlike the Java Virtual Machine, which interprets code at runtime, the Microsoft .NET runtime compiles in machine-native code, which provides much better performance at runtime.

Assemblies

In the past, developers shared libraries of code with one another through the creation of dynamic link libraries (DLLs). One developer could create a DLL, and other developers could reference and access this DLL in their own code. The only problem was that users were required to have this DLL registered on their machine in order for applications that reference it to work.

In the Microsoft .NET framework, developers can share libraries through managed components, which are known as *assemblies*. An assembly can exist as a single file or as a series of files. The major difference between assemblies and the legacy DLLs is that assemblies contain not only the code but also a manifest, which describes the assembly.

Manifests

Unlike traditional Component Object Model (COM) components that must be registered on a client’s machine, Microsoft .NET assemblies contain more than just the code

of the component. Additional information that describes the component is stored in the manifest, and therefore Microsoft .NET assemblies are often referred to as self-describing components. The manifest is much like a type library in that it describes the assembly and all its types, methods, and properties. This eliminates the need for assemblies to be registered in the system registry.

Benefits of Assemblies

In the past, one of the biggest headaches for developers was dealing with DLLs on client machines. If you needed to reference a library, you would have no guarantee that the client's machine had the proper version of the DLL. You could install the DLL along with your application, but that would not prevent another application from coming along and replacing it with another version of the DLL. This is because all DLLs are global to the client machine in which they reside. When you register a DLL, all applications on that system can access it.

A great benefit of having assemblies that are self-describing is that developers can create applications that have no impact on all other applications on the system. When you create an application that requires an assembly, that assembly is available only to that application. The assembly can be shared with other applications, but every application that uses the assembly must have the assembly copied to its `bin` directory.

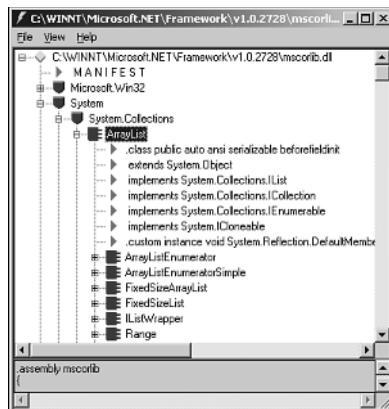
Having several copies of the same assembly is probably not a very good idea; if you make a change to the assembly, you must update it in all applications. Because of this, if you are going to create an assembly that you know will be used by many applications on a client machine, you can create a shared assembly, which like legacy DLLs, is available to the entire system.

Shared assemblies are structurally the same as private assemblies. The only difference is that shared assemblies must have a unique name and they are placed in the global assembly cache, which is located within the `WinNT\Assembly` directory.

Another benefit of self-describing assemblies is their ease of deployment. Because assemblies are not registered, an assembly can be installed by simply copying the compiled component to the client's machine.

The Intermediate Language (IL) Disassembler

The Microsoft .NET Software Development Kit (SDK) includes a tool called the IL Disassembler (`ildasm.exe`) which can be used to view the manifest and MSIL code for Microsoft .NET assemblies. Developers can use the tool to find out what types, methods, and constructors are contained within the assembly. The IL Disassembler is shown in Figure 2.1.

**Figure 2.1**

The IL Disassembler.

Garbage Collection in the Microsoft .NET Runtime

Languages such as C++ that do not implement garbage collection place the responsibility of managing an application's objects and memory in the hands of the developers. Other languages, such as Java and Visual Basic, provide a garbage collection system that handles the cleanup of the application's objects.

Like Java and Visual Basic, the Microsoft .NET runtime offers a garbage collection system. However, the Microsoft .NET garbage collection system varies quite a bit from the garbage collectors implemented in Java and Visual Basic. The following sections provide more details on the garbage collector built into the Microsoft .NET runtime.

The Microsoft .NET garbage collection system is based on the Mark and Compact algorithm. Basically, this system removes objects that have gone out of scope and compacts all the remaining objects to the beginning of the address space. In addition, to increase the performance of the garbage collector in Microsoft .NET, the collector implements the concept of generations; the system looks for freed-up space by looping through younger objects first. Both the Mark and Compact algorithm and the concept of generations are described in detail in the following sections. In addition, we will go through a number of scenarios to help explain the behavior of the garbage collector in Microsoft .NET.

The Mark and Compact Algorithm

When an application is started, a given amount of address space is allocated to the application. This space is called the *managed heap*. The managed heap is used to store the application's object references. When a new object needs to be instantiated, and if there is not enough room in the application's managed heap, the garbage collection

process runs. If objects have gone out of scope, the space is freed up and the new object is instantiated. However, if the garbage collector is unable to free up enough space, the application is able to instantiate the object, and an error occurs.

The Microsoft .NET garbage collector implements the Mark and Compact algorithm, which follows these steps:

1. The garbage collector generates a list of all the application's objects that are still in scope.
2. The garbage collector steps through the managed heap, looking for any objects that are not present in this list (which means they are out of scope and they can be collected).
3. The garbage collector marks the unreachable objects that are out of scope. The space these objects was taking up is therefore freed so that it can be used for other objects.
4. After the garbage collector has iterated through the managed heap, it moves the references to all the objects down to the beginning of the heap, filling all open space. All the object references are therefore stored contiguously in the heap, which makes the process of adding new object references to the stack much more efficient.

A simple case scenario makes this algorithm process clear. When the application is started, Objects A, B, C, D, and E are instantiated, and references to these objects are placed on the managed heap. This is shown in Figure 2.2.

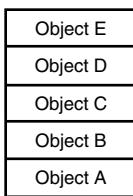


Figure 2.2

The managed heap before garbage collection.

Later in the life of the application, Object F needs to be instantiated. There is not enough space in the managed heap to allocate the object, and therefore garbage collection needs to be performed. Objects B, D, and E have gone out of scope since the last collection.

When the garbage collection is started, a list of all the objects that are still in scope is generated; the list contains Objects A and C. The algorithm then iterates through the managed heap, looking for objects that are not present in the list. These objects are marked as unreachable, and the managed heap is compacted. A reference to Object F is then placed on the heap. Figure 2.3 represents the managed heap after garbage collection has taken place.

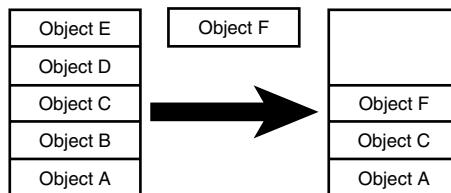


Figure 2.3

The managed heap after garbage collection.

Generations

As discussed in the previous section, the Microsoft .NET runtime stores all of an application's object references in a managed heap. When that heap is full, garbage collection occurs. The garbage collector marks all the objects that are out of scope as unreachable, frees the space, and compacts all the object references that are in scope to the beginning of the managed heap.

Because the Microsoft .NET runtime stores all the object references in contiguous space, it is very efficient to add new references to the managed heap. However, it could still be very expensive to step through the entire heap, looking for marked objects.

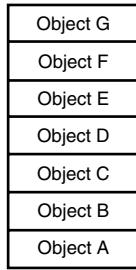
To help make this process more efficient, the Microsoft .NET runtime implements the concept of generations. The theory of generations assumes that the older an object is, the less likely it is to be out of scope during a garbage collection. Therefore, it is most efficient to check the newest objects first, and then check the older objects if space is still needed.

The Microsoft .NET garbage collector supports three generations: Generation 0, Generation 1, and Generation 2. Objects in Generation 0 are objects that have been created since the last garbage collection, objects in Generation 1 are objects that still remain after one garbage collection, and objects in Generation 2 are objects that are still in scope after multiple garbage collections.

When a garbage collection is performed, the garbage collector collects only objects in Generation 0. This increases the performance of the garbage collection because it is working with a smaller portion of the managed heap.

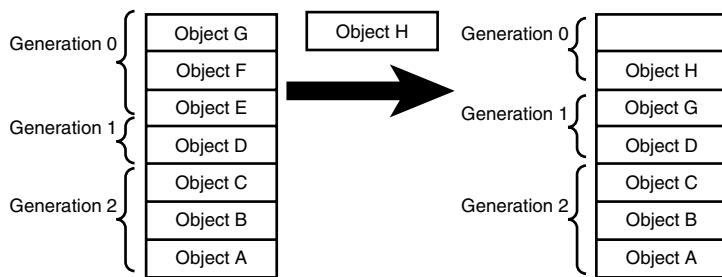
If there is still not enough space to allocate a new object after they have been collected from Generation 0, the garbage collector then moves on to Generation 1, and then if necessary, it moves to Generation 2 objects.

For example, consider an application that has the Objects A, B, and C in Generation 2, Object D in Generation 1, and Objects E, F, and G in Generation 0. Figure 2.4 represents the managed heap before the first garbage collection.

**Figure 2.4**

The managed heap before the first garbage collection.

The application attempts to allocate space to Object H, but it cannot, so a garbage collection is required. Objects E, F, and D have gone out of scope since before the first garbage collection. Figure 2.5 represents the managed heap after this garbage collection occurs.

**Figure 2.5**

The managed heap after the first garbage collection.

After the first garbage collection, Objects E and F were collected. Even though Object D had gone out of scope, the garbage collector only collected from Generation 0, so Object D was not removed. If more space had been required after the objects in Generation 0 had been collected, Object D, from Generation 1, would also have been collected.

Also notice that Object G is moved from Generation 0 to Generation 1 because it has survived one garbage collection. The garbage collector did not collect from Generation 1; therefore, Object D is still within Generation 1.

Manual Control of Garbage Collection

The Microsoft .NET garbage collector makes a collection only when an object cannot be instantiated until space is released. Because of this, there is no way to determine when an object is released from memory. When absolutely necessary, the Microsoft

.NET runtime does provide developers with the control over the collection process. This control is provided through use of the `GC` class in the `System` namespace, which offers the ability to force a collection by using the `Collect` method. In most cases, you will want to let the garbage collector handle things on its own, but there might be times when you want to do a garbage collection over all generations.

In addition to forcing a collection by using the `System.GC` class, you can determine the total amount of memory available to an application by using the `GetTotalMemory()` method, and you can determine the current generation of an object by using the `GetGeneration()` method. You can also mark an object as ineligible for garbage collection by using the `KeepAlive()` method.

CHAPTER 3

ASP.NET Web Forms

At the most basic level, an ASP.NET Web form looks remarkably like a regular HTML page. It is a text file that contains markup language that can be edited in a simple text editor such as notepad. However, an ASP.NET Web form has several distinguishing elements, which are discussed in this chapter:

- Web forms contain blocks of code that are processed on the server.
- The first time a Web form is requested, the entire page is compiled. Subsequent requests are served from this compiled page.
- Web forms can contain page directives, which are global settings for the page. Common page directives enable you to turn off session state and ViewState management.
- Web forms can contain both server-sides, including SSIs, which enable you to dynamically insert the contents of a file into your Web form, and user controls. User controls are object oriented and use a more programmatic approach to code encapsulation than SSIs.
- As mentioned previously, Web forms can also contain as much HTML and client-side script, such as JavaScript, as desired.

Server-Side Code Blocks

Code to be executed on the server is contained inside a set of `<script> </script>` tags, similar to the tags you would use to add client-side code to a Web page. In order for the page compiler to process the source code on the server, you must add a `runat="server"` attribute to the tag. Page logic, including the handling of events such as the `Page_Load()` event, is found in server-side code blocks rather than on the client side.

 **NOTE**

If the `runat="server"` attribute is not specified, then the tag will be passed to the client as is. This causes unpredictable results, depending on the tag in question. In the case of ASP.NET Web controls, the browser will most likely ignore the tag entirely. HTML controls will then be displayed on the client, but they will not be accessible to ViewState management or server-side validation.

Compiling ASP.NET Web Forms

As mentioned in Chapter 2, “The Common Language Runtime,” all source code on an ASP.NET Web form is dynamically compiled into an intermediate language and then compiled just-in-time (JIT) into native machine code. This is handled completely behind the scenes; you do not need to compile a Web form manually. Web forms are compiled for several reasons:

- Compiling the code in this way dramatically reduces the server’s response time and processor load. Because the server spends less time processing each request, it can process more requests. Therefore, by using the Microsoft.NET framework, you can get more “bang for the buck” from your Web servers.
- You can use strongly typed languages such as C#, Visual Basic.NET, JScript.NET, and managed C++ code for Web development on a Web form itself. This forces you to use proper coding techniques and helps eliminate the use of unstructured code that mars many standard Active Server Pages (ASP) applications.
- The page compiler finds many errors that might be overlooked until runtime in an interpreted language such as VBScript in ASP.

The Intermediate Language (IL) that is generated in ASP.NET is human readable. You can learn a lot about the page processor by making changes to a page and seeing how the IL is affected.

Not just the server-side code is compiled into the IL; every element of the page is compiled, including the Hypertext Markup Language (HTML). HTML is placed into a text literal control and then added to the controls collection for the page.

Consider the simple Web form example in Listing 3.1. This page displays a message, using a label Web control. The text of the label control is specified on the server side. Listing 3.2 shows the IL code generated from the code in Listing 3.1. A complete discussion of the conversion of a Web form to IL is beyond the scope of this book. However, as you can see from Listing 3.1, even a very simple Web form can translate into some relatively intimidating code.

Listing 3.1 A Very Simple Web Form

```
<HTML>
```

```
<HEAD>
<script language="C#" runat="server" >
```

Listing 3.1 continued

```

    void Page_Load(Object Source, EventArgs E)
    {
        msg.Text = "Hello World!";
    }
</script>
</HEAD>
<BODY>
<asp:Label id=msg runat="server"></asp:Label>
<%myerror%>
</BODY>
</HTML>
```

Listing 3.2 Automatically Generated Intermediate Language Code

```

Line 11:     namespace ASP {
Line 12:         using System;
Line 13:         using System.Collections;
Line 14:         using System.Collections.Specialized;
Line 15:         using System.Configuration;
Line 16:         using System.Text;
Line 17:         using System.Text.RegularExpressions;
Line 18:         using System.Web;
Line 19:         using System.Web.Caching;
Line 20:         using System.Web.SessionState;
Line 21:         using System.Web.Security;
Line 22:         using System.Web.UI;
Line 23:         using System.Web.UI.WebControls;
Line 24:         using System.Web.UI.HtmlControls;
Line 25:
Line 26:
Line 27:     public class myPage_aspx : System.Web.UI.Page,
System.Web.SessionState.IRequiresSessionState {
Line 28:
Line 29:         private static System.Web.UI.AutomaticHandlerMethodInfos
                __autoHandlers;
Line 30:
Line 31:
Line 32:         #line 12 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 33:         protected System.Web.UI.WebControls.Label msg;
Line 34:
Line 35:         #line 1000000 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 36:
Line 37:         private static bool __initialized = false;
Line 38:
Line 39:         private static System.Collections.ArrayList
                __fileDependencies;
Line 40:
Line 41:         #line 4 "c:\inetpub\wwwroot\Book\mypage.aspx"
```

Listing 3.2 continued

```
Line 43:  
Line 44:     void Page_Load(Object Source, EventArgs E)  
Line 45:     {  
Line 46:         msg.Text = "Hello World!";  
Line 47:     }  
Line 48:  
Line 49: #line 1000000 "c:\inetpub\wwwroot\Book\mypage.aspx"  
Line 50:  
Line 51:     public myPage_aspx() {  
Line 52:         if ((ASP.myPage_aspx.__initialized == false)) {  
Line 53:             System.Collections.ArrayList dependencies = new  
System.Collections.ArrayList();  
Line 54:             dependencies.Add  
("c:\inetpub\wwwroot\Book\mypage.aspx");  
Line 55:             ASP.myPage_aspx.__fileDependencies = dependencies;  
Line 56:             ASP.myPage_aspx.__initialized = true;  
Line 57:         }  
Line 58:     }  
Line 59:  
Line 60:     protected override  
System.Web.UI.AutomaticHandlerMethodInfos AutoHandlers {  
Line 61:         get {  
Line 62:             return ASP.myPage_aspx.__autoHandlers;  
Line 63:         }  
Line 64:         set {  
Line 65:             ASP.myPage_aspx.__autoHandlers = value;  
Line 66:         }  
Line 67:     }  
Line 68:  
Line 69:     protected System.Web.HttpApplication ApplicationInstance {  
Line 70:         get {  
Line 71:             return ((System.Web.HttpApplication)  
(this.Context.ApplicationInstance));  
Line 72:         }  
Line 73:     }  
Line 74:  
Line 75:     public override string TemplateSourceDirectory {  
Line 76:         get {  
Line 77:             return "/Book";  
Line 78:         }  
Line 79:     }  
Line 80:  
Line 81:     public override void InstantiateIn(System.Web.UI.Control  
control) {  
Line 82:         this.__BuildControlTree(control);  
Line 83:     }  
Line 84:
```

Listing 3.2 continued

```

Line 85:     private System.Web.UI.Control __BuildControlmsg() {
Line 86:         System.Web.UI.WebControls.Label __ctrl;
Line 87:
Line 88:         #line 12 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 89:         __ctrl = new System.Web.UI.WebControls.Label();
Line 90:
Line 91:         #line 1000000 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 92:         msg = __ctrl;
Line 93:
Line 94:         #line 12 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 95:         __ctrl.ID = "msg";
Line 96:
Line 97:         #line 1000000 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 98:         return __ctrl;
Line 99:     }
Line 100:
Line 101:    private void __BuildControlTree(System.Web.UI.Control
Line 102:                                         __ctrl) {
Line 103:        #line 1 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 104:        this.__BuildControlmsg();
Line 105:
Line 106:        #line 1000000 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 107:
Line 108:        #line 1 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 109:        ((System.Web.UI.IParserAccessor)
Line 110:             (__ctrl)).AddParsedSubObject(this.msg);
Line 111:
Line 112:        #line 1000000 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 113:        __ctrl.SetRenderMethodDelegate(new
Line 114:            System.Web.UI.RenderMethod(this.__Render__control1));
Line 115:
Line 116:    private void __Render__control1
Line 117:        (System.Web.UI.HtmlTextWriter __output,
Line 118:         System.Web.UI.Control parameterContainer) {
Line 119:        System.Web.UI.Control Container;
Line 120:        Container = parameterContainer;
Line 121:        __output.Write("<HTML>\r\n\r\n<HEAD>\r\n\t");
Line 122:        __output.Write("\r\n</HEAD>\r\n<BODY>\r\n      ");
Line 123:
Line 124:        #line 12 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 125:        parameterContainer.Controls[0].RenderControl(__output);
Line 126:        #line 1000000 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 127:        __output.Write("\r\n      ");
Line 128:
```

Listing 3.2 continued

```
Line 127:         #line 13 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 128:         myerror
Line 129:
Line 130:         #line 1000000 "c:\inetpub\wwwroot\Book\mypage.aspx"
Line 131:             __output.Write("\r\n</BODY>\r\n</HTML>");
Line 132:         }
Line 133:
Line 134:     protected override void FrameworkInitialize() {
Line 135:         this.FileDependencies =
Line 136:             ASP.myPage_aspx.__fileDependencies;
Line 137:         this.EnableViewStateMac = true;
Line 138:
Line 139:     public override int GetTypeHashCode() {
Line 140:         return -1269963154;
Line 141:     }
Line 142: }
Line 143: }
```

ASP.NET Web Form Events

From the moment a user first requests an ASP.NET Web Form to the moment the server actually sends the response, quite a few events and processes occur. A few of the most important ones are described in the following sections.

A significant number of events occur between the moment a client issues a page request and the moment that the server sends the response. Many of these events will be transparent to you as the developer, and they will involve the compiler, the pre-processors, and the various collections that make up the .NET framework. However, you will need to understand some events because you will likely come across them in your coding escapades. Figure 3.1 shows the events in a life cycle of an ASP.NET request.

Page Directives

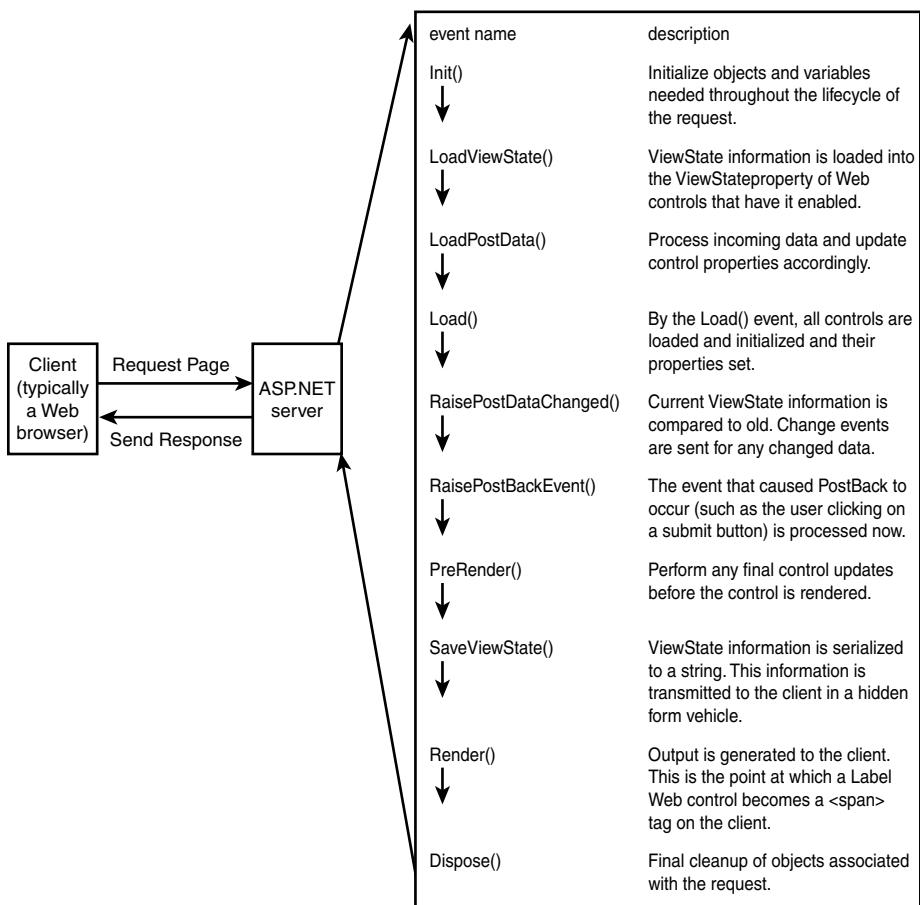
Page directives enable you to set optional metasettings for the compiler when it processes the ASP.NET Web form. You can use page directives to set the default language for the ASP.NET Web form, enable or disable session state, choose to buffer the contents of the page before sending to the client, and make a host of other changes.

Page directives must be located at the very top of an ASP.NET Web Form, before any other code. A page directive is always contained inside `<% %>` tags and always begins with the `@` character. Here is an example of the syntax used for a page directive:

```
<%@ directive.property = value %>
```

For example, to turn off session state for a page, you use the following page directive:

```
<%@ Page EnableSessionState="false"%>
```

**Figure 3.1**

The life cycle of an ASP.NET request.

Server-Side Includes

SSIs offer a way to insert the contents of a file into an ASP.NET Web form. Because an SSI is inserted before the page is compiled to IL, you can use an SSI to insert blocks of source code into a page. You can use SSIs to add a copyright notice to the bottom of a page, include a common set of functions, or include any other sort of static content.

The code for an SSI looks like this:

```
<!-- #include PathType = FileName -->
```

PathType can be set to either File or Virtual, depending on the type of path being referenced:

- File sets a relative path from the directory in which the ASP.NET Web Form is located. It is important to note that the file location cannot be in a directory higher on the file tree than the current directory. This prevents a possible breach in a site's security.
- Virtual enables you to include a file from a virtual path from a virtual directory on the Web server.

TIP

You should not use SSIs to encapsulate site functionality such as site navigation or common site text. ASP.NET offers a solution for these types of situations, called *user controls*, which are explored in the following section.

User Controls

User controls provide a way to encapsulate site functionality in ASP.NET. Because user controls are self-contained in standalone files, you can think of them as a “black box.” Variables are explicitly passed to user controls by using public properties. Similarly, for user control methods to be accessed outside the scope of the user control, they must be declared as public. If the user control contains HTML or Web Form information, then it will be produced inline, where the user control was included in the calling form. In this way, a user control is like any other server control.

User controls are self-contained entities, and they support fragment caching. By using a user control and fragment caching, it is possible to cache the output of a single piece of functionality on a page. For instance, if a page in an application contains a menu and highly personalized user content, it would be difficult to send the entire page to the output cache because of the high number of potential permutations of the personalized content. For instance, if a page welcomes a user by name, it would be difficult to cache the entire page since a separate copy of the page would need to be stored in memory for each user name. Encapsulating certain pieces of the page, such as the menu, into a user control and then turning on fragment caching for the user control increases the overall performance of the page.

Because user controls can be programmatically manipulated, several developers can use user controls to work on the design of an ASP.NET application simultaneously. Because each user control is self-contained, each developer can work on his or her part of the puzzle and not worry too much about interfering with someone else’s code.

Visual Studio and the Code Behind Method

Currently, there are two major paradigms for designing ASP.NET Web Forms:

- One method is to insert server-side code on a page that contains all the HTML and client-side code. This method of building pages should be familiar to ASP developers.

- Because ASP.NET Web Forms are compiled, you can place presentation logic such as HTML and any Web controls in one file and any server-side code (such as C# or Visual Basic.NET) in a second file. At compile time, the two pages are compiled as one entity. This method of designing ASP.NET Web Forms is called *code behind*.

Although many users might be reluctant to use this method, the code behind method offers some benefits. Presentation code such as HTML is neatly separated from any page logic being performed. This makes it easy for a development group in charge of graphics and layout to work independently of a group in charge of back-end development. Visual Studio.NET uses this code behind method of development. In fact, it doesn't support the traditional in-line code methodology at all.

If you have done significant development in previous versions of ASP, you might find code behind difficult to work with at first, especially when combined with all the other new features in ASP.NET. It's definitely worth learning to use, however, Visual Studio.NET impressively shortens development time.

CHAPTER 4

ASP.NET Controls

ASP.NET controls are a set of server-side blocks of code that generate Hypertext Markup Language (HTML) or perform other specific actions when placed on an ASP.NET page. There are several different kinds of ASP.NET controls including HTML controls, Web controls, validation controls, and list controls. Each different type of control serves a different purpose.

ASP.NET controls appear visually to be the same as other XML tags with one exception: For the page compiler to recognize that a control is to be processed on the server side, the `runat="server"` attribute must be included in the control's tag. Any tag without this property will just pass through the processor and reach the client untouched. Most browsers simply ignore extraneous tags. For instance, here is an example of a valid HTML control that will process successfully:

```
<input type=button runat="server">
```

HTML controls are the most basic type of control. They look like standard HTML tags. However, because they are processed on the server, they can take advantage of postback, server-side validation, ViewState management, and other capabilities provided by the Microsoft .NET framework.

Web controls are designed to provide a layer of abstraction to HTML. The various Web controls enable you to control Web page layout. Each Web control maps to a different HTML tag and thus provides a wide range of functionality. Several of the Web controls enable you to create input forms. Like the HTML controls, Web controls can be used with postback, server-side validation, and ViewState management.

Validation controls are Web controls that have a very specific mission: to provide server-side validation for input controls. Using different validation controls, you can easily compare values, check to make sure an input value falls within a range of values, and ensure that an input is not left blank. You can also easily create your own custom validation control if none of the default validation controls suits your specific purpose.

List controls are designed to bind to data (such as a dataset retrieved from a database) and display this data on the page. There are three types of list controls: Repeater, DataGrid, and DataList, each of which provides unique formatting capabilities and control.

TIP

If a control will not appear or function correctly on the ASP.NET page, check to make sure you have not forgotten the `runat="server"` property. If you have forgotten to include this property, there will not be an error; the tag will either not display or worse: display as a standard HTML element and cause you to wonder why it won't respond to server side events.

ASP.NET Controls Generate HTML

ASP.NET controls provide a layer of abstraction to HTML on the page by dynamically generating the required HTML. The type and quantity of HTML generated varies greatly from control to control, but this section provides some basic examples. Listings 4.1 though 4.3 show examples of some simple controls and the HTML that they generate.

Listing 4.1 The ListBox Web Control

```
<asp:listbox id="listbox1" VisibleItems=5 Style="Width:100" runat=server>
  <asp:Listitem>One</asp:Listitem>
  <asp:Listitem>Two</asp:Listitem>
  <asp:Listitem>Three</asp:Listitem>
</asp:listbox>
```

Listing 4.2 The ListBox Web Control Rendered to the Page

```
<select name="listbox1" id="listbox1" VisibleItems="5" size="5"
  style="Width:100">
  <option value="One">One</option>
  <option value="Two">Two</option>
  <option value="Three">Three</option>
</select>
```

Listing 4.3 The DataGrid List Control

```
<ASP:DataGrid id="MyDataGrid" runat="server"
    Width="500"
    BackColor="#ccccff"
    BorderColor="black"
    ShowFooter="false"
    CellPadding=3
    CellSpacing="0"
    Font-Name="Verdana"
    Font-Size="8pt"
    HeaderStyle-BackColor="#aaaaaa"
    EnableViewState="false"
/>
```

As previously mentioned, the List controls bind to a data source (data retrieved from a SQL Server database, for example) and display the results on the page. The DataGrid in Listing 4.3 generates HTML like that shown in Listing 4.4. The actual HTML will vary, depending on the data source passed to the list control.

Listing 4.4 The DataGrid List Control As Rendered to The Page

```
<table cellspacing="0" cellpadding="3" rules="all" bordercolor="Black"
border="1" style="background-color:#CCCCFF; border-color:Black; font-family:
Verdana; font-size:8pt; width:500px; border-collapse:collapse;">
    <tr style="background-color:#AAAAAA;">
        <td>
            Country
        </td><td>
            LastName
        </td><td>
            FirstName
        </td><td>
            ShippedDate
        </td><td>
            OrderID
        </td><td>
            SaleAmount
        </td>
    </tr><tr>
        <td>
            UK
        </td><td>
            Suyama
        </td><td>
            Michael
        </td><td>
            7/10/1996 12:00:00 AM
    </tr>
</table>
```

Listing 4.4 continued

```
</td><td>
    10249
</td><td>
    1863.4
</td>
</tr>
</table>
```

The amount of HTML generated in Listings 4.2 and 4.4 two examples might seem trivial. However, consider that the `Calendar` Web control generates several orders of magnitude more HTML than the Web controls above. The HTML wrapped up inside an ASP.NET control is tested for various browsers to ensure compatibility and robustness before being released. This keeps you from reinventing the wheel each time you need to implement a certain functionality.

ViewState Management

One of the most common difficulties that developers face in creating a Web application is managing state information. *Maintaining state* is the process of keeping track of variable and control information from one section of the application to another section.

The main problem (and benefit in some cases) with standard HTML is that it is a disconnected protocol. The browser requests data from the server, the data is served, the connection is closed, and the browser displays the information to the user. By using only standard HTML, it is nearly impossible to keep track of user information between separate pages in the same application.

Over the years, ASP developers have discovered many innovative ways to manually manage state information, such as by using hidden input fields and session variables. These solutions often require a large amount of programming overhead to implement and maintain.

The days of manually maintaining state are gone, thanks to the introduction of the Microsoft .NET framework. ViewState management enables you to automatically cause state information to persist between requests.

In standard ASP/HTML, causing a user's input to persist across form postings or even failed form postings could require copious amounts of client-side code because the user's previous entries must be loaded (usually from a database) and then the various entries must be pushed back into the appropriate form inputs. The Microsoft .NET framework automatically handles all this, which reduces the development overhead considerably.

Implementing ViewState Management

ViewState management is implemented in a rather ingenious way; the information needed to maintain state is sent to the client, along with the Hypertext Transfer Protocol

(HTTP) response. This removes much of the potential overhead of storing the state information for each request on the server.

The code that ViewState management generates to maintain state looks much like the code in Listing 4.5. For the most part, in order to prevent a potential breach of security, the actual contents of the state information is not human readable. Listing 4.5 shows a typical block of ViewState information, taken from the IBuySpy application (www.ibuyspy.com).

Listing 4.5 ViewState Management Implementation

```
<INPUT type="hidden" name="__VIEWSTATE" value="a0z1479145297_a0z_hz5z7x_
a0z_hz5z3x_
a0z_hz5z1x_a0za0za0zhzDataKeys_lzx_
ItemCount_5z4x_DataSourceItemCount_5z4xx_x_____x_hz5z0x_a0z_hz5z4x_a0z_
hz5z5x_a0za0zhz0_$89.99x_x_x_5z4x_a0za0zhz0_$89.99x_x_x_5z2x_a0za0zhz0_
SHADE01x_x_x_5z1x_a0za0zhz0_Ultra Violet Attack Defenderx_x_x_5z0x_a0z_
hz5z1x_a0za0zhz0_379x_x_xxxxxx_5z3x_a0z_hz5z5x_
a0za0zhz0_$1.99x_x_x_5z4x_a0za0zhz0_$1.99x_x_x_5z2x_a0za0zhz0_LKARCKTx_x_x_
5z1x_a0za0zhz0_Pocket Protector Rocket Packx_x_x_5z0x_a0z_
hz5z1x_a0za0zhz0_373x_x_xxxxxx_5z2x_a0z_hz5z5x_a0za0zhz0_
$129.99x_x_x_5z4x_a0za0zhz0_$129.99x_x_x_5z2x_a0za0zhz0_WOWPENx_x_x_5z1x_
a0za0zhz0_Mighty Mighty Penx_x_x_5z0x_a0z_hz5z1x_a0za0zhz0_371x_x_xxxxxx_5z1x_
a0z_hz5z5x_a0za0zhz0_
$69.99x_x_x_5z4x_a0za0zhz0_$69.99x_x_x_5z2x_a0za0zhz0_1MOR4MEx_x_x_5z1x_
a0za0zhz0_Cocktail Party Palx_x_x_5z0x_a0z_hz5z1x_a0za0zhz0_386x_x_
xxxxxxxxxxxxxx_azupdate_MyList:
ctrl13:Remove_MyList:ctrl15:Remove_checkout_MyList:ctrl12:Remove_MyList:ctrl14:
Removexx\n0\Te\xt">
```

Turning Off ViewState Management to Increase Performance

Because ViewState management sends extra information—about 4KB worth—to the client and requires extra server processing to implement, it should be turned off if it is not needed in a particular application or Web form. Even though 4KB may not sound like much additional overhead, if there are, for example, 500 hits in one minute, the amount of information served increases by 2MB. (Keep in mind that 4KB and 500 hits are both conservative estimates.)

ViewState management can be turned off on a per-control basis or for an entire form at a time, using the `EnableViewState` attribute. Simply add `EnableViewState=False` to a control to disable ViewState management for that control, or add it to the top of a page to turn off ViewState management for all controls on the page.

For example, to turn off ViewState management for a `TextBox` Web control, you can use the following code:

```
<asp:TextBox runat="server" EnableViewState="False"></asp:TextBox>
```

HTML Controls

HTML controls are regular HTML tags with the `runat="server"` attribute specified. The page compiler thus processes the tag on the server enabling the benefits of the Microsoft .NET framework, such as ViewState management and validation. The following lines show how some typical HTML controls appear when placed on an ASP.NET page:

```
<form id=form1 runat="server"></form>
```

and

```
<a href="mylink.asp" id=href1 runat="server"></a>
```

Because the object model for HTML controls is nearly the same as for standard HTML, the learning curve is minimal for just about any Web developer. Furthermore, the controls can fully utilize ViewState management and can allow a developer who is new to the Microsoft .NET framework to create a robust application quickly.

HTML controls can fully participate in postback and thus save you time because you do not have to manually keep track of and manage input values.

HTML controls can fully participate in the event life cycle of an ASP.NET page. Thus, in the `Page_Load()` event, if you give an HTML control an ID, you can programmatically manipulate it and modify the exposed properties.

Web Controls

Web controls provide an additional layer of abstraction over HTML controls. Rather than mapping almost directly to HTML like the HTML controls, Web controls provide a logical and consistent object model and then generate proper HTML to the browser.

You place Web controls on an ASP.NET page by using the following syntax:

```
<asp:Classname id="IdentifyingID" Property="value" runat="server"/>
```

or

```
<asp:Classname id="IdentifyingID" Property="value" runat="server">  
</asp:Classname>
```

A commonly used Web control is the `Label` Web control, which is generally used to render a message to the user, as in the following example:

```
<asp:Label id="label1" Text="Hello World!" runat="server"></asp:Label>
```

Benefits of Using Web Controls

Even though Web controls are at first glance more difficult to understand than HTML controls, they offer many more benefits and functionality.

Web controls automatically detect the version and type of browser requesting data and send a tailored response for that particular browser. This can prevent you from wasting a lot of time having to code separate user interface versions for various Web browsers. Also, the HTML that the Web controls generate has been already tested and tweaked.

Because Web controls encapsulate and generate potentially large amounts of HTML, you can spend less time coding and tweaking minor user interface points and instead spend time on more high-level architectural issues.

Web controls have full access to ViewState information and can thus easily cause control values to persist by using postback. Like HTML controls, Web controls are processed on the server. The page compiler ignores any Web control that does not have the `runat="server"` property specified.

Though most of the built-in web controls appear relatively simple and generate a specific HTML element, the controls can be as complex as you want to make them. Consider the Calendar Web control. It consists of several hundred lines of code and when placed on a Web form, it renders a fully functional calendar.

The Web Control Base Class

All Web controls are derived from the same base class. This ensures that the object model remains consistent across various controls which offers several benefits. For instance, the base class provides a standard object model for accessing style information for the controls and offers other functionality which needs to be present in all controls. For example, it is impossible to explicitly control the tab order of a form in standard HTML without rearranging elements. Even with the addition of a client-side scripting language such as JavaScript, controlling the tab order of form elements is very difficult to achieve and maintain. The Web control base class makes implementing features such as a tab index possible. By using the Web control `TabIndex` property, you can control the tab order—that is, the order in which the cursor moves through the form elements as the user presses the Tab key—of the form elements on an ASP.NET page.

The base class also provides a way to disable Web controls by exposing the `Enabled` property. You can easily enable or disable (that is, gray out) HTML entities without writing a single line of client-side script. As in the case of tab order, disabling Web controls is very difficult to achieve with standard HTML and ASP.

Event Handling

ASP.NET offers several page events. The following sections describe the page events that are most commonly used.

The Page_Load() Event

You use the `Page_Load()` event to perform form processing logic and to perform any data binding and other initialization that might be needed by the controls and other items on a page. You have full access to all control and ViewState information. As a general rule, almost all pages have code inside the `Page_Load()` event.

The On_Click() Event

The `On_Click()` event pertains to the button Web control and certain other form elements. You can place whatever code you desire inside a button's `On_Click()` event. This enables you to control what happens when a user clicks the button.

Rich Controls

As previously mentioned, Web controls are not limited to generating a simple block of HTML. You can place as much functionality as desired into a Web control. At a basic level, rich controls are overloaded Web controls with a much larger degree of complexity. Two rich controls that are included with the Microsoft .NET platform Software Development Kit (SDK) are the `AdRotator` and the `Calendar` rich controls.

You may be familiar with the ActiveX controls that would read a configuration file and display banner ads across the top of a Web page. The `AdRotator` control provides the same functionality as these ActiveX controls. `AdRotator` configuration is performed in a human-readable XML file.

The `Calendar` control is an extremely useful control that is used in gathering date information. Placing the `Calendar` control on an ASP.NET page inside a server-side form will render a calendar that the user can use to select a date. The `Calendar` control exposes a wealth of formatting properties and is thus extremely customizable.

Custom Controls

It is possible to create your own controls in C#, Visual Basic.NET, managed C++, or any of the other languages in the Microsoft .NET framework. Because custom controls are compiled and neatly encapsulate functionality, there is no doubt that there will be a large market for third-party ASP.NET controls just as there is a large market today for ActiveX controls.

CHAPTER 5

List Controls

Working with lists of data is a very common task for developers. Product listings, items in a shopping cart, customer listings, and collections of links are just a few of many different types of lists that developers work with on a daily basis. Despite the fact that working with lists has been common for some time, in the past there was not a very good way to efficiently work with lists. ASP.NET makes working with lists easy by providing list controls.

The ASP.NET list controls are a set of controls that were created to help developers work with lists of data. The list controls provide a number of rich features that allow developers to handle complicated tasks, which may have taken pages of code to perform in ASP, in only a few lines of code. This dramatic improvement makes the list controls one of the most immediate benefits of working with ASP.NET.

This chapter provides an overview of the list controls. It describes three of the List controls—`Repeater`, `DataList`, and `DataGridView`—and discusses the many benefits of working with list controls. This chapter also talks about how list controls work, and Chapter 11, “Working with ASP.NET List Controls,” describes how to implement list controls.

Types of List Controls

This chapter describes three list controls: `Repeater`, `DataList`, and `DataGridView`. Each of the controls is geared toward solving problems in different scenarios. This section provides an overview of each of these list controls, to give you a better understanding of when to use each of these controls.

Repeater

The Repeater control is the simplest of the list controls; it is a barebones control for working with lists of data. The Repeater control renders items in a list based on developer-defined templates (templates are described in detail in chapter 11). You use the Repeater control when you do not need the special features that are supported by the DataList and DataGrid controls. For instance, the Repeater control would be the ideal control if you simply wanted to display a list of links to the user. An example of displaying a list of links in the Repeater control is shown in Figure 5.1.



Figure 5.1

An example of the Repeater list control.

DataList

The DataList control is similar to the Repeater control, but it is a more advanced list control. DataList, like Repeater, renders data based on a developer-defined template. For each item in the list, the DataList control generates the HTML from the appropriate template. A product listing is a great example of using a DataList control. An example of using the DataList control to create a product listing is shown in Figure 5.2.

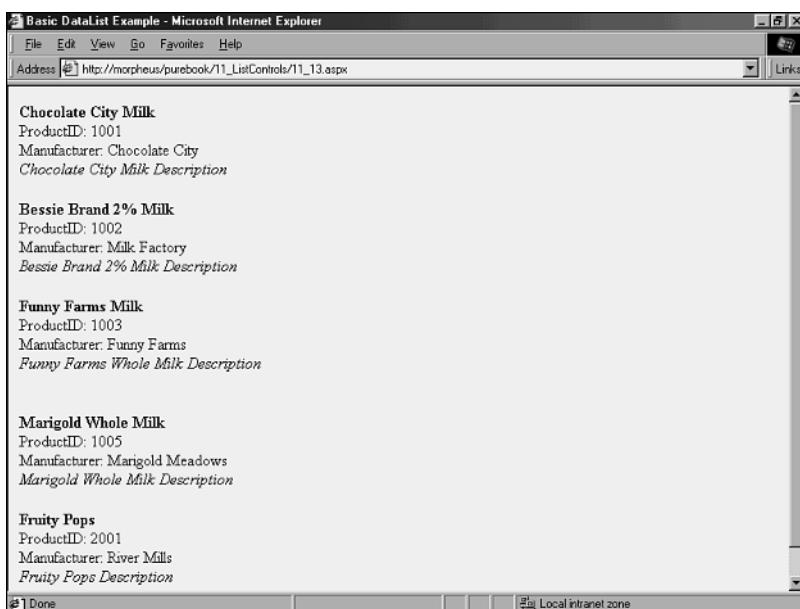


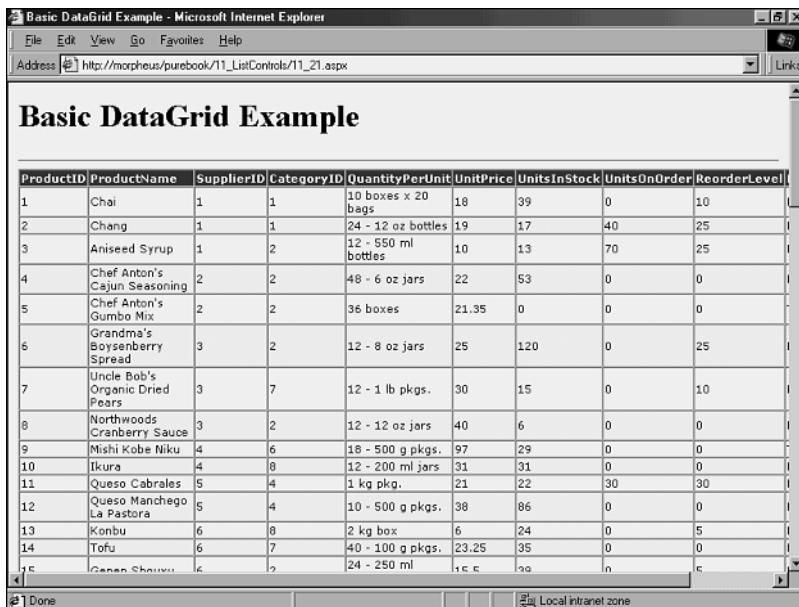
Figure 5.2

An example of the *DataList* control.

The major difference between the Repeater and the *DataList* controls is that the *DataList* control supports a number of rich features that the Repeater control doesn't support. For example, the *DataList* control provides a number of style properties that you can use to easily change the appearance of the *DataList* control. In addition, *DataList* supports some common list operations such as the ability to display the list items across multiple columns, select items in a list, and edit items in a list.

DataGrid

The *DataGrid* control implements many of the same rich features that the *DataList* control supports. However, unlike the Repeater and *DataList* controls, *DataGrid* is used to display lists of data that have multiple columns. An ideal use of *DataGrid* would be to build a shopping cart. The cart could contain a number of records, each representing a product. Each product could have a number of pieces of information, such as the quantity of the product, the name of the product, the unit cost of the product, and the calculated cost of product. An example of using the *DataGrid* control is shown in Figure 5.3.

**Figure 5.3**

The DataGrid control.

Of all the list controls, the DataGrid control supports by far the most features. One of the most useful features is automatic column generation. For example, say you have a data source that contains three fields: Product Name, Description, and Unit Cost. You want to display a grid that shows all three fields for each record in the data source. In order to do this, you simply need to bind the data source to the DataGrid control. DataGrid then automatically determines what fields are in the data source, and for each one, it generates a new column in the grid.

Other features that the DataGrid control include editing data inline, sorting data by columns, spreading data over a number of pages, and applying style properties. These features are discussed in the next section, “Benefits of Using List Controls.”

Benefits of Using List Controls

The ASP.NET list controls help you work with lists of data. They provide a number of benefits, including simplified development, browser independence, and built-in support for common list operations. This section describes each of these benefits.

Simplified Development

One of the most obvious benefits of list controls is the amount of time and energy they save. Tasks that took pages of code in ASP take only a few lines in ASP.NET, thanks to list controls.

Consider the common task of displaying a table that shows a list of customers and their contact information. In ASP, you would retrieve a data source containing the list from the database, and you would then loop through each record in the data source and for each item, you would generate HTML code. By using list controls, you can do the same task much more quickly. You place a `DataGridView` control on the page, create a data source that contains the list, and you bind the data source to the control. The list control automatically goes through each item in the list and generates the table for you. This is just one sample of the ways list controls can simplify the development involved in working with lists of data.

Browser Independence

One of the growing difficulties with Web application development is support for multiple browsers. As the number of browsers is increasing, so is the variance between the features that they all support. Therefore, you must be very careful to develop sites so that all browsers that are directed to your site can view it properly.

As with all the other ASP.NET server controls, the list controls render data based on the browser with which it is being displayed. This allows developers to write code that can be viewed by many different types of clients quickly and easily.

Built-in Support for Common Operations

When you work with lists of data, a number of operations come up frequently. The ASP.NET list controls support the most common of these operations—such as selecting, paging, sorting, displaying data across multiple columns, and editing—which are described in the following sections.

Table 5.1 summarizes the features that are supported by the different ASP.NET list controls.

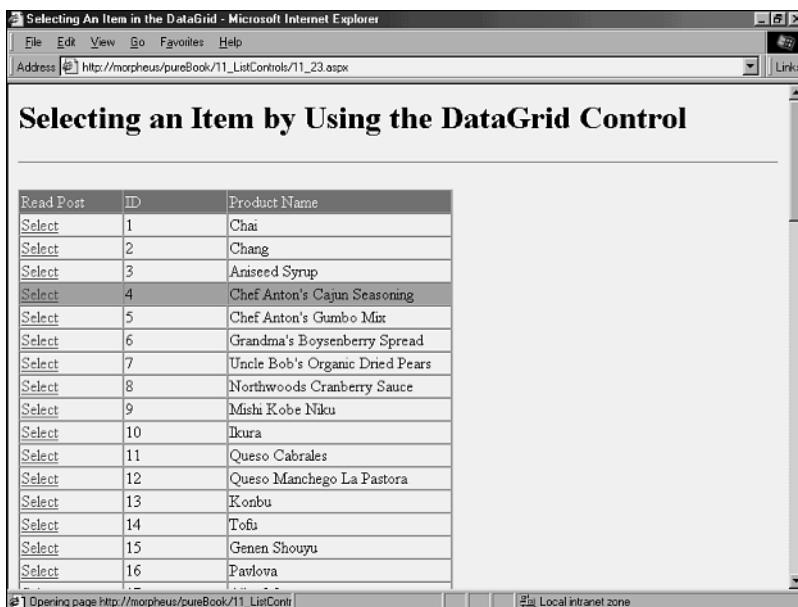
Table 5.1 ASP.NET List Control Feature Support

Feature	Repeater	DataList	DataGrid
Editing	No	Yes	Yes
Paging	No	No	Yes
Sorting	No	No	Yes
Selecting	No	Yes	Yes
Multiple-column display	No	Yes	No
Styles	No	Yes	Yes
Templates	Yes	Yes	Yes (within column definitions)
Automatic column generation	No	No	Yes

Selecting

Often when working with lists, you need to select a given item from a list. The ASP.NET `DataList` and `DataGridView` list controls both support the ability to select items from a list. Both controls keep track of the currently selected item and allow you to

display them differently than the rest of the items using a different template. An example of selecting an item with the **DataGrid** control is shown in Figure 5.4. More information on how to select items in a **DownList** and **DataGrid** control, see chapter 11.

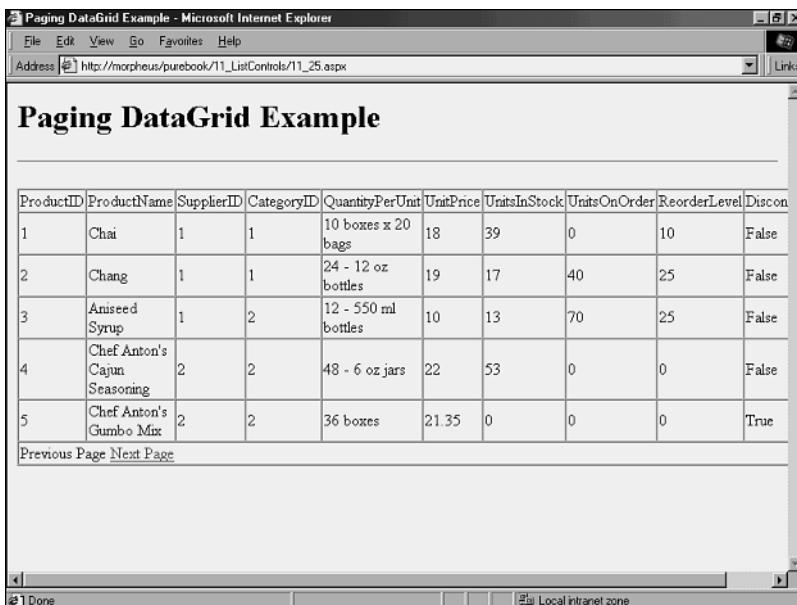
**Figure 5.4**

Selecting an item with the **DataGrid** control.

Paging

In some situations, you need to display large lists of data. It is not user-friendly to display the entire list to the user at one time because that forces the user to read and scroll through a lot of information. To make the display of large lists more friendly to users, you can display the data across a number of pages (for example, 10 items per page). For example, you could use this feature to display lists of links generated by a search. Figure 5.5 shows an example of using the **DataGrid** control to display data over multiple pages. For more information about how to implement paging with the **DataGrid** control, refer to Chapter 11.

You can use the **DataGrid** to activate paging by simply setting a few of its properties. You can choose from a couple paging interfaces, or you can create your own paging interface by tapping into the **DataGrid** control's paging features.

**Figure 5.5**

Paging data with the DataGrid control.

Sorting

Another way to make the display of large lists of data friendlier is to allow users to sort the data. For example, if a user clicks on a column header, the data can be sorted by the column. With sorting, users can easily locate the information they need by browsing through a list in order.

The DataGrid control has built-in support for sorting. When sorting is activated in a data grid, the column headers become hyperlinks by default. When the user clicks one of the column header links, the DataGrid control sorts the data by that column. You can disable sorting for those columns that you do not want to support sorting. An example of sorting with the DataGrid is shown in Figure 5.6. See Chapter 11 for more information on how to implement sorting with the DataGrid control.

Displaying Data Across Multiple Columns

Often there are times when you need to display lists of data across two or more columns. (An example of this is the product listing shown in Figure 5.7.) In the past, displaying data across multiple columns was a rather tedious task because you had to determine which of the items to put in Column 1, which to put in Column 2, and so on.

46 Chapter 5: List Controls

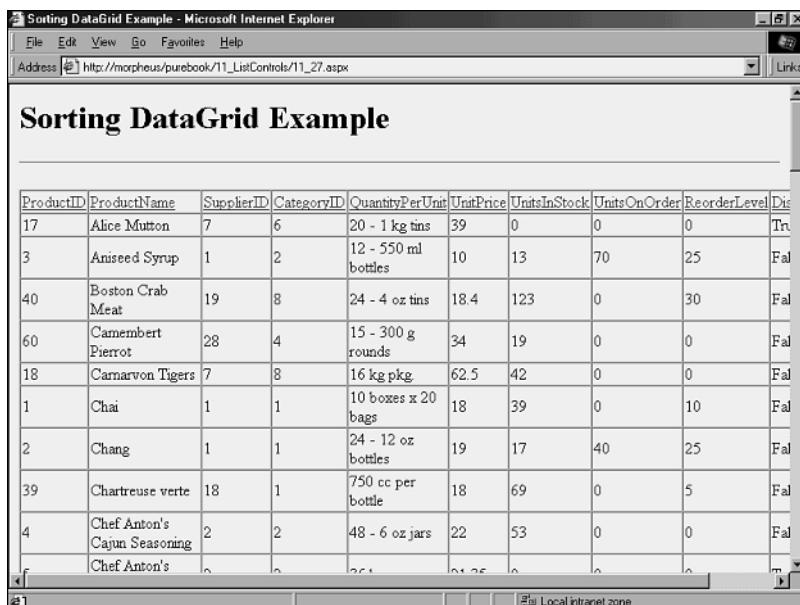


Figure 5.6

Enabling sorting of data with the DataGrid control.

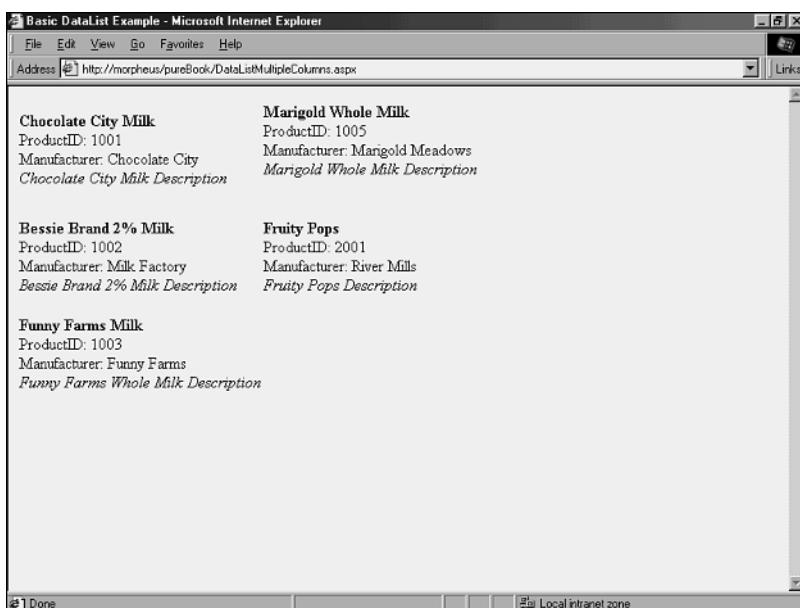


Figure 5.7

Displaying data across multiple columns with the DataList control.

The **DownList** control supports the ability to display lists across a number of columns. By setting two properties for the **DownList** control, the items in a list will automatically be spanned across any specified number of columns, either horizontally or vertically.

Editing

Another of the many features supported by the ASP.NET list controls is the ability to easily edit items inline. Both the **DataGrid** and **DownList** controls support editing of list items by allowing you to define an edit template, which then defines the way the items are rendered when editing. . A common use for this feature would be to insert editable web controls into the List Controls, which allows users to edit the data in the list. An example of inserting editable text boxes in a **DataGrid** control is shown in Figure 5.8.

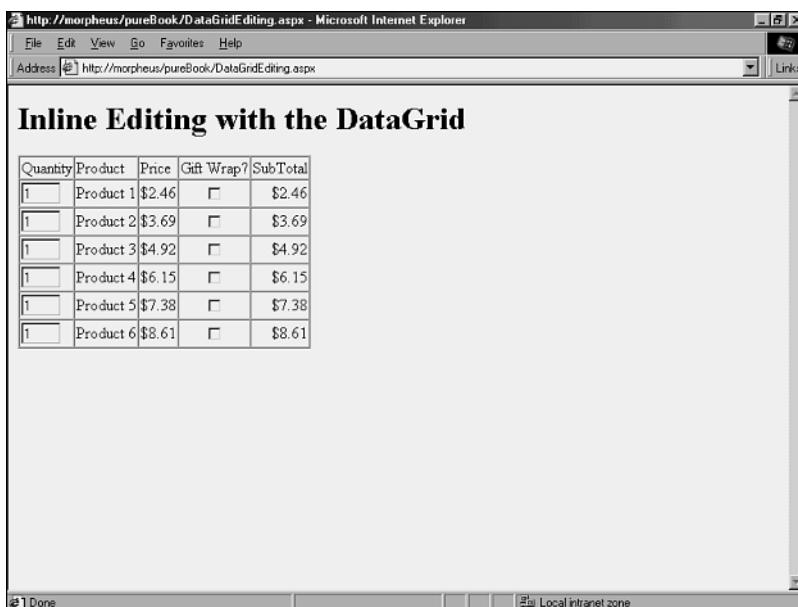


Figure 5.8

Inline editing, using the DataGrid control.

How the List Controls Work

To fully understand the list controls, it is important to understand how they function. This section discusses the type of collections that can be bound to the list controls as well as the way the controls work with their underlying lists.

Data Binding to List Controls

You use list controls to display lists of data from a bound data source. After you retrieve a list of data in your data source, you must have a way for the list controls to

read the data. To do this, you need to bind the data source to the list control by setting the control's `DataSource` property.

To bind a list to a list control, the control must understand how to read the list from the data source. To make this possible, the list controls restrict the types of data sources that can be bound to them to those that implement the `IEnumerable` interface. Some of the most popular types that implement `IEnumerable` are the `DataView`, `HashTable`, and `ArrayList` controls. For more information on data binding to the list controls, see chapter 11.

When a data source is bound to a list control, the control doesn't actually read the data source until the `.DataBind()` method is invoked. When this method is called, the list control steps through the data source, and for each record in the list, the control adds another Item to its item collection. After the item collection is populated, the list control displays the list according to the defined templates for the `Repeater` and `DataList` controls or displays the list in the grid for the `DataGrid` control.

CHAPTER 6

Validating User Input

What Are Validation Controls?

In Web development, user-input validation is the process of ensuring that the user has entered legitimate values into input fields. Though the required values may change from page to page, the type of validation performed does not change much.

In standard HTML, validation is performed mainly in one of two ways. One way is to use client-side code to check input values before they are even sent to the server. This method of validation has several benefits. There is no round trip to the server unless the values are correct, and if the values are not correct, a friendly message can be displayed to the user, stating the nature of the error. However, this method does not work if the user's browser does not support client-side scripting or if the user has client-side scripting turned off for security reasons. Also, this method requires you to think on three separate levels: client-side script, server-side script, and HTML.

Another way to perform validation involves checking for errors on the server after the form has been submitted. This method works very reliably, but it has a few drawbacks. The user discovers an error only after a potentially time-consuming round-trip to the server. Also, poorly planned validation performed in this manner usually stops after the first incorrect value, leaving the user to make several more round-trips to the server to successfully post the form.

Many real-world applications are a combination of these two methods. Validation implemented in this manner is fairly robust and offers a variety of features. However, because validation involving both client-side and server-side code is inserted into two layers of the application, it could consume

large amounts of your time and energy, both in the creation and maintenance of the code.

The validation controls in ASP.NET are geared toward providing a more logical and consistent validation model. Rather than having to write custom JavaScript methods to handle client-side validation and then write similar functions in a form handler, in case the client either does not support JavaScript or has it disabled, with ASP.NET it is possible to place one line of code on a page and let the Microsoft .NET framework do most of the work for you.

The validation controls are instantiated on a Web form in the same manner as any other Web control. Here is an example of a required field validation control:

```
<asp:RequiredFieldValidator runat="server"  
ControlToValidate="SomeOtherControl" id="myValidator">
```

As with all other Web controls, this tag is processed on the server. Then, any client-side code that is necessary to validate the field is automatically generated by the Microsoft .NET framework and sent to the client.

Benefits of Using Validation Controls

The major advantage of using validation controls is that they reduce the number of lines of code in an application. Validation controls provide a number of other benefits as well. Validation controls offer a consistent way to validate data. In older versions of ASP, methods used to validate user input could vary wildly, based on the whims of individual developers. The following sections detail other benefits of using validation controls.

Using Uplevel Versus Downlevel HTML Browsers

Uplevel browsers are defined as Internet Explorer version 4.x and up as well as Netscape 6 and later versions. ASP.NET takes advantage of the rich feature set in these browsers and provides validation that requires no round-trip to the server. The necessary client-side JavaScript used to perform client-side validation is automatically generated and sent to the client. If an invalid entry is submitted, the browser detects the error on the client side and displays an error message to the user without contacting the server. This reduces server load and also improves the end user's experience.

Downlevel browsers do not support JavaScript well enough to fully participate in the validation process. However, downlevel browsers can take advantage of the benefits of validation controls. In order for validation to occur, the browser must contact the server, where the input is validated and then a response is sent back to the browser. The validation control will still contain the same error message, but validation will appear to be more sluggish, because a round-trip is required to the server.

Eliminating Client-Side Scripting to Validate Input Data

Keeping track of server-side code, client-side code, and presentation logic in traditional applications can be difficult at times. Because ASP.NET validation controls are coded

and manipulated on the server side, you do not need to create large amounts of client-side script to perform validation. This reduces a layer of complexity for you as the Web developer.

Providing Consistent Error Reporting

If a field fails validation, validation controls are able to display an error message to the user. The error message is placed in-line in the HTML where the validation control was placed on the Web form. Additionally, whenever a control fails validation, the `Page.IsValid` property is set to False. This property can be checked before processing a form. In addition, the `ValidationSummary` object contains all errors from all validation controls for a particular page. This enables you to conveniently display all problems with form input to the user.

Separating Control

Validation controls are separate from the controls they are validating. Conceivably, the validation framework could have been built into Web controls themselves through a set of common methods and properties in the Web control base class. However, validation error messages are placed wherever the validation control is placed on the Web form. This gives you a lot of flexibility in designing the user interface. There are a number of possibilities for control placement. The validation controls can all be placed at the top or bottom of the Web form, to keep error messages grouped together, or they can be placed in-line next to the control they validate, to cue the user to the input that needs attention.

Types of Validation Controls

Five built-in types of validation controls available in the ASP.NET Framework: `RequiredField`, `Range`, `Comparison`, `RegularExpression`, and `Custom`.

The `RegularExpressionValidator` control provides rich functionality that suits the needs of almost all validation scenarios. For times when this is not the case, the `Custom` control can be used; it enables you to completely customize the validation logic and cover all other cases. However, as shown in the following sections, each validation control has unique strengths.

The RequiredFieldValidator Control

The `RequiredFieldValidator` control is primarily used to make sure that an input field is not left blank. It can also be used to make sure that the user does not leave the default value for a field unchanged (for example, when formatting information for the field is provided in the field, such as MM-DD-YYYY for a date input). For more information on implementing the `RequiredFieldValidator` (and the other validation controls), see Chapter 12, “Working with ASP.NET Validation Controls.”

The RangeValidator Control

The `RangeValidator` control ensures that an input field is within a range of values, as defined by the `Maximum` and `Minimum` properties of this validation control. In other

words, if you have a `RangeValidator` control with 0 as `Minimum` and 10 as `Maximum`, then any value between 0 and 10 will pass validation.

The CompareValidator Control

The `CompareValidator` control enables you to compare two values and check for equality, greater-than, less-than, and so on. You can also use this control to ensure that a user's input is of a certain data type.

An ideal use of this control is to compare the common password and confirm password fields that are prevalent on online registration screens.

The RegularExpressionValidator Control

The `RegularExpressionValidator` control uses a Perl 5-based regular expression language to determine whether a field passes validation. Regular expressions are a sort of shorthand notation for processing strings. A complete discussion of the regular expression language is beyond the scope of this book.

The `RegularExpressionValidator` control is used primarily to ensure that the form of a user's input meets certain criteria. For instance, it could easily be used to check for properly formed telephone numbers (three numbers, three numbers, four numbers), Social Security numbers (three numbers, two numbers, 4 numbers), e-mail addresses (nonblank text, @ symbol, text for domain, ., and a top-level domain), or any other string that must be entered with a consistent format.

The CustomValidator Control

The `CustomValidator` control is used when intricate or granular validation needs to occur. This control enables you to define your own specific criteria for validation. The `CustomValidator` control points to two validation functions: one on the server side and the other on the client side. These functions can contain any logic you want in order to define validation. The functions return `True` when the criteria passes validation and `False` when validation fails.

Displaying Errors

Displaying errors with validation controls is very straightforward. The following sections detail the appearance and placement of error messages.

Error Formatting

The format of an error message can easily be customized as much as desired. You can use control properties such as `BackColor`, `BorderStyle`, and `BorderWidth` to directly format the appearance of an error message. In addition, you can apply a cascading style sheet (CSS) style to an error message, by using the `CssStyle` property. This gives you the ability to modify almost every aspect of the appearance of the error message. A terrific reference on CSS properties can be found at msdn.microsoft.com/workshop/author/css/reference/attributes.asp. For more information on formatting errors, see Chapter 12.

Validation Control Placement

If the value of a control doesn't validate, the error message is displayed wherever the validation control is placed on the Web form. This gives you very fine control over the placement of error messages.

CHAPTER 7

Understanding Data Access with ADO.NET

ADO.NET is a set of namespaces that enables you to connect to a data source (such as Microsoft SQL Server or any valid OLE DB data source) and work with data. ADO.NET was designed to offer a consistent object model for working with data from almost any data source.

ADO.NET was also designed to be as similar as possible to ADO. It should not take long for a developer who has worked with ADO to learn how to use ADO.NET. The `Command` and `Connection` objects in particular should make you feel right at home.

Benefits of ADO.NET

ADO.NET offers several benefits over ADO. Because data in ADO.NET is represented as eXtensible Markup Language (XML), ADO.NET is inherently interoperable. In addition, because ADO.NET uses disconnected copies of data, it is more scalable and offers better performance than ADO.

Interoperability

In ADO, transmitting a result set to a remote application was potentially a difficult and at times daunting task. If the remote application was a Microsoft-based solution as well, then marshalling the data as an ADO recordset was only as difficult as configuring the two servers and any firewalls in between for the data transmission.

However, if the remote application resided on a non-Microsoft platform such as Linux, then the direct marshalling of the ADO recordset object was not possible. The recordset had to be

transformed to a commonly recognized format (typically XML) and then transmitted to the remote application.

In ADO.NET, data is represented interally and marshaled as XML automatically. This enables a Microsoft .NET application that uses ADO.NET to easily share data with any other development platform that understands XML.

ADO.NET can also connect to a wide range of data sources by using OLE DB. OLE DB provides connectivity to many data types, including Microsoft Access, Microsoft Excel, Oracle databases, and ASCII text files. The `System.Data.OleDbClient` namespace provides OLE DB supprt. In addition, `System.Data.SqlClient` provides a managed provider for Microsoft SQL Server version 7.0 and above.

Scalability and Performance

Historically, the number of open database connections has been a major bottleneck in application design. That is, each open connection to the database requires a fair amount of system resources. Thus, a good practice to follow in writing code to retrieve data is to open a connection to the database as late as possible in your code and close the connection as quickly as possible.

In contrast to ADO.NET, ADO maintains open connections to the database for as long as the recordset exists in memory. The recordset could be disconnected from the database fairly easily, but this behavior did not occur by default.

By default, data in ADO.NET is disconnected from the database. This means that the ADO.NET classes retrieve data and then immediately close the connection to the database. This feature enables you to build efficient and scalable applications; you can work with disconnected recordsets automatically, with no additional code.

The ADO.NET Object Model

Most of the ADO.NET classes exist in three namespaces. `System.Data` contains the `DataTable` and `DataSet` classes, which provide an in-memory database for ADO.NET. The `System.Data.SqlClient` namespace provides the classes necessary to work with Microsoft SQL Server versions 7.0 and up. The `System.Data.OleDb` namespace provides the classes necessary to work with OLE DB data sources. The `Connection`, `Command`, and `DataReader` objects exist in each of these last two namespaces, and closely mirror each other in functionality. For more information about namespaces, please see Chapter 14.

Figure 7.1 shows how the various ADO.NET objects relate to one another.

For more information on the ADO.NET objects discussed in this chapter, refer to Chapter 13, “Data Access with ADO.NET,” or Chapter 20, “`System.Data.SqlClient` Reference.”

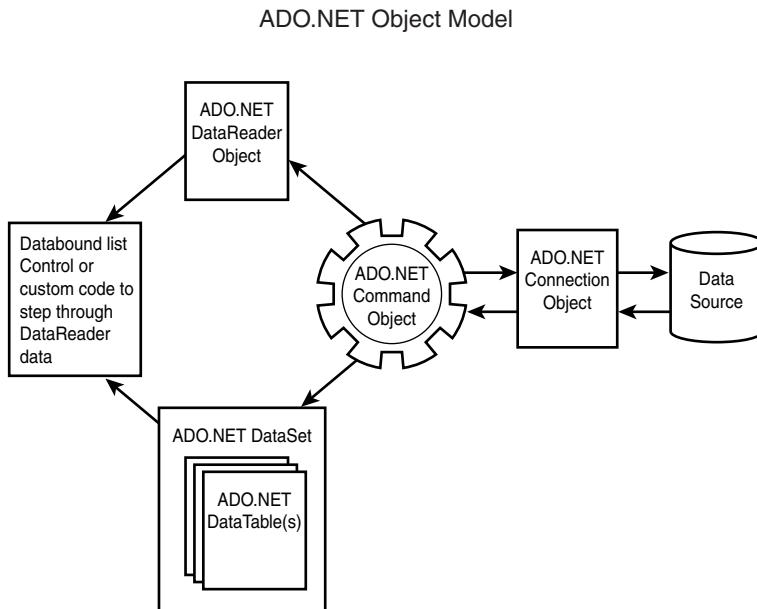


Figure 7.1

A look at the ADO.NET object model.

The DataTable Class

A **DataTable** object is a collection of rows of data. The **DataTable** object is roughly analogous to a recordset in ADO. However, because **DataTable** objects store more than just data, they can better be compared to database tables in relational databases. **DataTable** objects can contain information about the data's primary key, if any, as well as constraint and relationship information (provided by the **DataSet** object, described in the next section).

The DataSet Class

A **DataSet** object is a collection of **DataTable** objects and all the database relations between those **DataTable** objects.

An ADO recordset is basically a grid of data. An ADO.NET **DataSet** object is a rich in-memory database representation of data. Because the **DataSet** object stores information about the data as well as the data itself, you can join and manipulate data from heterogeneous sources. You can pull data from a mainframe into **DataTable** Object A and data from a Microsoft SQL server into **DataTable** Object B and then work with A and B with as much flexibility as if you were working with two tables in Microsoft SQL Server.

The `DataSet` class also contains methods to read and write XML, which enables you to represent a `DataSet` as XML at any point.

The Connection Class

The ADO.NET `Connection` object provides a connection to a data source, and it is also used to initiate a database transaction. The ADO.NET `Connection` object is very similar in appearance to its ancestor, the ADO `Connection` object.

There are two connection objects built into the Microsoft .NET framework: `SqlConnection` and `OleDbConnection`. The `SqlConnection` object is found in `System.Data.SqlClient` and is used to make a connection to a Microsoft SQL Server version 7.0 or higher database. As you might have guessed, the `OleDbConnection` object is found in the `System.Data.OleDb` namespace and handles connections to OLE DB data sources. These two objects have nearly identical properties and methods.

The Command Class

The ADO.NET `Command` object enables you to execute commands against a data source. Using the `Command` object, you can retrieve, delete, insert, or update data, and you can use any other valid SQL statement on your data source. The `Command` object also contains a collection of parameters that are used to pass information to stored procedures in the database.

The DataReader Class

The `DataReader` object is used to retrieve a read-only, forward-only stream of data from a data source. Because only one row of data is ever in memory at a time, using the `DataReader` object is a very efficient way to retrieve data, particularly large amounts of data.

CHAPTER 8

Overview of Web Services

As the Internet and distributed computing become increasingly prevalent, the need for applications to access services on a remote machine securely also increases. Accessing remote services has been added to older development platforms (for instance, Distributed Component Object Model [DCOM]) almost as an afterthought. However, Web services are fully integrated into the Microsoft .NET framework.

What Are Web Services?

In the broadest sense, a Web service is a method or set of methods that are publicly available to remote applications over Hypertext Transfer Protocol (HTTP). The Microsoft .NET framework enables you to create and deploy these Web services quickly and easily by providing all the “plumbing” necessary to receive, parse, and respond to a request from a client.

Distributed Computing

Currently, one of the biggest challenges in software development is designing distributed applications. Before the advent of Web services, developers often had difficulty calling a method or function on a remote machine and receiving a response, even if the machines were on the same network.

The Distributed Component Object Model (DCOM) and remote procedure calls (RPCs) enable you to make programming calls to a remote server. However, these protocols are object-model specific, and with them, you must be concerned about binary compatibility and DLL versions. DCOM also requires additional registry settings be created to enable a component to be accessed by a remote system. Additionally, using DCOM across the Internet can quickly become a nightmare because you must try to manage firewall configurations at both the client and server locations.

Web services solve all these problems. As long as the remote server can be reached via HTTP (and no advanced packet filtering specifically prevents .asmx files from being requested), Web services will function. In fact, it is possible to browse directly to the Web service in a Web browser and view publicly accessible methods as well as the Web Service Description Language (WSDL) contract. The WSDL contract defines the public interface of the Web service.

SOAP for Communication

Simple Object Access Protocol (SOAP) is an industry-standard XML-based data transmission format. Because with SOAP the message itself is XML based, very complicated data and commands can be transferred with ease. This enables XML to describe data and represent intricate data schemas.

Web services can use SOAP (over HTTP) for message delivery, in addition to the standard HTTP GET and HTTP POST methods. The specific syntax of each of these message transmission types is covered in Chapter 15, “Building Web Services.” However, you probably do not need to be concerned with the message syntax. This knowledge might come in handy to debug a problematic Web Service, but various utilities shield you from the message complexity.

Why Use Web Services?

Web services offer several benefits. Web services operate over HTTP, so they have nearly universal access through firewalls. Because the interface of a Web service is all that needs to be defined publicly, the actual implementation is hidden. Also, the entire set of built-in Microsoft .NET framework classes are at your disposal, enabling you to create very powerful functionality.

Standard Web Ports

Web services operate over port 80, the default port for HTTP traffic. Network administrators may be concerned about this because they are responsible for preventing unwanted attacks by hackers. However, Web services were designed with security in mind from the very beginning. It is no more dangerous to expose a Web service to the world than it is to expose your Web server to the world.

Black Box Functionality

The implementation of a Web service is completely shielded from the consuming application (that is, the application that is invoking the Web service). The consuming application only knows the rules for interfacing with the Web service (for example, arguments, returns). Thus, proprietary code and calculations remain hidden.

Site Functionality Exposure

It is possible to expose some or all of your site functionality in the form of a set of Web services. For instance, an e-commerce site could implement a Web service that enables remote clients to check on the status of an order. A warehouse could implement a real-time inventory checker that could instantly report an out-of-stock message to a client.

A scientific research firm could offer Web services that provide extremely processor-intensive, proprietary calculations. The possibilities are endless.

Programming Model Flexibility

You can create Web services by using any of the languages supported by the .NET framework. This includes, but is not limited to, C#, Visual Basic.NET, and JScript.

Class Library Support

Web services offer the same access to all the namespaces and classes as a standard Web form. This enables you to create very powerful Web services quickly.

Security

Because Web Services typically operate over HTTP, securing a Web service is done in exactly the same manner as securing a Web page. For more information on the types of security available in the .NET framework, see Chapter 17, “Securing and Deploying an ASP.NET Application.”

Improved End User Experience

Web services have the potential to change the face of software development. Because processing and data can be housed and accessed on a remote server, client applications can become more powerful. For instance, an ultralightweight client such as a mobile phone with a Web browser could access a Web service designed to search a national phone directory by last name.

Web services do not just retrieve data; they can also be used to accept data and store it. This opens the door to enabling some or all user data to being stored online, where it can be backed up. It is easy to integrate these Internet services into any type of Microsoft .NET application and enrich the end user experience.

Exposing Web Services

Exposing a Web service is a straightforward process. First, create a file with the file extension .asmx and place it anywhere in the virtual directory of a .NET application. Then, place the code for a public Web service into the file and save. The next time this .asmx file is requested, the Web service will be compiled and ready to be consumed.

Listings 8.1 and 8.2 show an extremely simple Web service. The syntax is explained in depth in Chapter 15. You can see that this Web service accepts two integer values and returns their sum. When the code in Listings 8.1 or 8.2 is compiled, the Microsoft .NET framework builds all the code to accept and parse the SOAP request that is calling the Web service and also builds the SOAP response.

Listing 8.1 A Simple Web Service (C#)

```
<%@ WebService Language="C#" Class="TrivialAddition"%>
```

```
using System.Web.Services;
```

Listing 8.1 continued

```
public class TrivialAddition : WebService {  
    [ WebMethod ]  
    public int Add(int operand1, int operand2) {  
  
        return operand1 + operand2;  
    }  
}
```

Listing 8.2 A Simple Web Service (Visual Basic.NET)

```
<%@ WebService Language="VB" Class="TrivialAddition"%>  
  
imports System.Web.Services  
  
Public Class TrivialAddition  
  
    public function <WebMethod ()>  
        Add(operand1 as Integer, operand2 as Integer) as Integer  
  
            Add = operand1 + operand2  
  
    End function  
  
End Class
```

Now that you have seen a simple Web service, it is time to look at ways to find Web services on a target server so that they can be used in a remote client application.

Discovering Web Services

The code in Listings 8.1 and 8.2 creates a publicly available Web service. Even though the service is publicly available, a client would need to know the exact URL of the Web service in order use it. This sort of security by obscurity may be desirable in the case of private Web services, but what if you have a Web service you would like clients to be able to reach easily?

To enable programmatic discovery of a Web service, a discovery file is used. A discovery file is an XML-based document that contains information about the Web services available in a specific application. A discovery file must be created and linked to the default page of your application to enable programmatic discovery.

Consuming Web Services

When a client application accesses a Web service method, the client application is said to be *consuming* a Web service. When a Web service is consumed, a SOAP request is sent over HTTP to the Web server. The XML in the SOAP request is parsed, and the data is sent to the Web service. The Web service then processes the request and formulates a SOAP response to send back to the calling application. When the consuming

application receives the SOAP response from the server, it must parse the XML in the SOAP response to read the data.

If the consuming application is a Microsoft .NET framework application, you will not need to perform any manual processing or parsing of XML. A built-in utility in the Microsoft .NET framework, `WebServiceUtil`, builds all the “plumbing” necessary to receive the SOAP response. Also, as previously mentioned, you don’t need to be concerned with the processing of the SOAP request to the Web service because Microsoft .NET handles it for you.

NOTE

It is possible to consume Microsoft .NET framework Web services from other platforms. A Web service requires a valid SOAP request in order to process data. Likewise, the consuming application only needs to have the ability to parse the XML in a SOAP response. Therefore, Web services are completely platform independent. For more information on processing SOAP requests in Visual Basic 6.0, you can download the Microsoft SOAP Toolkit from www.microsoft.com/downloads.

WSDL

In order to consume a Web service, the user only needs to know the location of the Web service. The user can programmatically discover or manually give the location in the form of a URL (as explained in the section “Discovering Web Services”), and the method signatures of those publicly available services. That is, WSDL contracts define all method arguments and define the output of the methods, if any.

The WSDL page for the Web service contains all the information needed to start consuming the Web service. When you have discovered the URL for the Web Service, you can navigate directly to the WSDL contract by appending `?WSDL` to the end of the URL (for example, `http://localhost/MyWebService.asmx?WSDL`).

When the WSDL contract of a Web service is located, all you need to do in order to use the Web service in a consuming application is to create a proxy class.

Proxy Classes

Because the methods and arguments of the Web service are unknown to a consuming client, any attempt to compile a remote application accessing the Web service will generate an error at compile time.

A proxy class contains a mirror image of all the method definitions in a Web service (but not the method contents). It basically lets the application know that such a function exists, and it creates the request to the service and sends the response back to your application.

You are probably eager to start coding all these proxy classes by hand, but the utility `WebServiceUtil` that is included with the .NET framework automatically creates a proxy class. You only need to provide the URL for the Web service, and `WebServiceUtil` creates the proxy class for you.

Calling a Web Service

Other than the logistics associated with network traffic, after the proxy class for the Web service is created, accessing the Web service methods is no different from calling a local method: You simply reference the proxy class in the application and then access the methods directly, with no special code required. For more information on the implementation of Web services, see Chapter 15.

PART II

DEVELOPING WITH ASP.NET

- 9 Building ASP.NET Pages with HTML and Web Controls
- 10 Encapsulating ASP.NET Page Functionality with User Controls
- 11 Working with ASP.NET List Controls
- 12 Working with ASP.NET Validation Controls
- 13 Data Access with ADO.NET
- 14 Building Components for ASP.NET
- 15 Building Web Services
- 16 Configuring and Optimizing an ASP.NET Application
- 17 Securing and Deploying an ASP.NET Application



CHAPTER 9

Building ASP.NET Pages with HTML and Web Controls

Web controls are a core technology in the development of ASP.NET pages. Much like design-time controls of days gone by, Web controls offer the Web developer a simple object interface for coding ASP.NET pages. However, Web controls offer a more consistent object and event model than design-time controls.

Web controls provide a level of abstraction between the Web developer and the Hypertext Markup Language (HTML) that the requesting client actually receives. The Web developer does not need to be concerned with minor HTML differences across browsers. The HTML that Web controls generate is automatically adjusted so that it appears correctly across various browsers.

A Technical Overview of Web Controls

You place a Web control on an ASP.NET page as an eXtensible Markup Language (XML) tag by using code like the following:

```
<asp:Classname id="IdentifyingID" Property="value"  
➥runat="server"/>
```

or

```
<asp:Classname id="IdentifyingID" Property="value"  
runat="server">  
</asp:Classname>
```

The `asp:` prefix defines the namespace of the Web control. `Classname` is the actual name of the Web control. The `id` attribute allows you to programmatically manipulate the control in server-side blocks of code.

When processing a page, the page compiler instantiates each control that is marked with the `runat` attribute set to `server` and then adds it to the control hierarchy tree in the proper position. This enables the Web control to participate in such things as view state management and validation.

You can set the properties of a Web control within the tag by using `Property="value"` syntax, as shown in the earlier example. Properties that the page compiler does not understand are simply passed through to the generated HTML (that is, garbage in, garbage out [GIGO]). In this manner, you can still access HTML attributes that the Web controls do not explicitly support. This ensures that you are not limited by the object model of the Web control.

For instance, consider the following `Label` Web control:

```
<asp:Label napkin="true" id="label1" runat="server">
```

When this control is compiled and run, it will generate the following HTML:

```
<span napkin="true" id="label1">
```

Core Web Controls

This section lists the core ASP.NET Web controls. These include general Web controls that map to specific HTML tags and Web controls that are used with form elements.

NOTE

Although validation controls are Web controls, they are very unique and specialized and are thus covered separately, in Chapter 12, “Working with ASP.NET Validation Controls.”

General Web Controls

The general Web controls are easy to understand because they map to basic HTML tags. That is, a particular Web control generates a corresponding HTML tag and attributes. For instance, the `Label` Web control is used to display a message on the page. To do this, it uses a `` HTML tag. Thus, the `Label` Web control is said to map to ``.

The following Web controls are covered in this section:

- `Label`—This is used to display a message on the page.
- `HyperLink`—This is used to place a hyperlink on the page.
- `Image`—This is used to place an image on the page.
- `Panel`—This is used to place a `<div>` HTML element on the page.
- `Table`—This is used to place a table on the page.

The Label Control

Table 9.1 The Label Control at a Glance

Property	String Text
----------	-------------

The purpose of the Label Web control is to display text. It generates a HTML tag on the client. The Text property of the Label control is used to define the contents of the span. Listings 9.1 and 9.2 demonstrate how the Label control is used. Figure 9.1 shows how Listings 9.1 and 9.2 appear in Microsoft Internet Explorer 5.5.

Listing 9.1 An Example of a Label Control (C#)

```
<%@ Page Language="VB" %>

<%@ Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void button1_OnClick(Object Source, EventArgs E)
        {
            label1.Text = textbox1.Text;
        }
    </script>

</HEAD>
<BODY>

<h2>Label Control Example</h2>
<hr>
    HTML tag>
<form runat="server" id=form1 name=form1>
    <ASP:Label id=label1
        style="background-color:#DDDDDD;font-weight:bold"
        runat=server>Hi There!
    </ASP:Label><br><br>
    <ASP:TextBox id=textbox1 Text="Enter some text" runat=server/>
    <asp:Button id=button1 Text="submit" OnClick="button1_OnClick"
        runat="server"/>
</form>

</BODY>
</HTML>
```

Listing 9.2 An Example of a Label Control (Visual Basic.NET)

```
<%@ Page Language="VB" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >
        Sub button1_OnClick(sender as Object, e as EventArgs)
            label1.Text = textbox1.Text HTML tag>
        End Sub
    </script>

</HEAD>
<BODY>

<h2>Label Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <ASP:Label id=label1
        style="background-color:#DDDDDD;font-weight:bold"
        runat=server>Hi There!
    </ASP:Label><br><br>

    <ASP:TextBox id=textbox1 Text="Enter some text" runat=server/>
    <asp:Button id=button1 Text="submit" OnClick="button1_OnClick"
        runat="server"/>
</form>

</BODY>
</HTML>
```

With this simple `Label` Web control, the user can enter a value into the provided `TextBox` Web control. (See the section “The `TextBox` Control” later in this chapter for more information on the `TextBox` control.) When the user clicks the submit button, the `button1_OnClick()` event transfers the text from the `textbox` to the `label`.

`Label` controls are useful for debugging purposes because they can be used to quickly display the contents of a string anywhere on the page. The `Label` Web control is also commonly used to display messages to the user during the handling of forms (for instance, when validation of a form field fails).

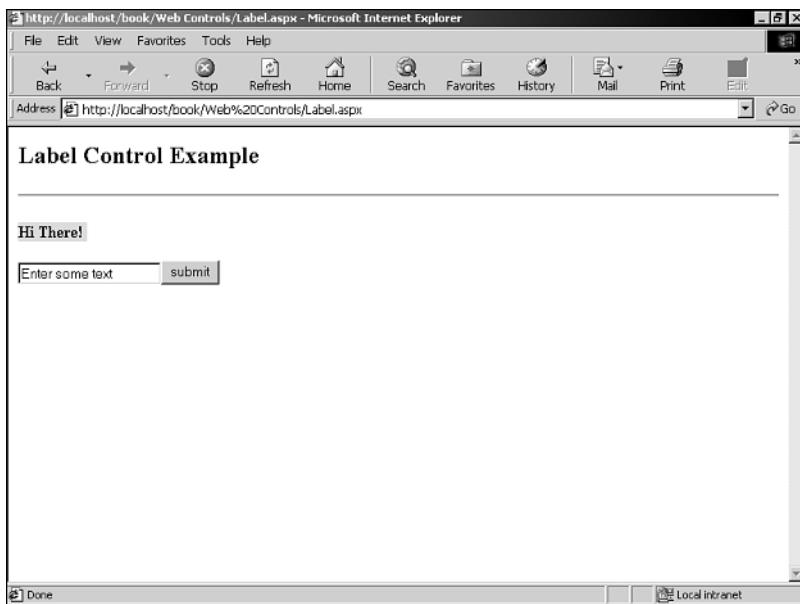


Figure 9.1

A Label Web control.

The HyperLink Control

Table 9.2 The HyperLink Control at a Glance

Properties	String Text String NavigateUrl String ImageUrl String Target
------------	---

The HyperLink Web control is used to create a hyperlink on a page. Specifically, it generates an anchor tag (that is, `<a>`), which is sent to the client. Listing 9.3 shows an example of how the HyperLink Web control is used. Figure 9.2 shows how Listing 9.3 appears in Microsoft Internet Explorer 5.5.

Listing 9.3 An Example of a HyperLink Control

```
<%@ Page Language="C#" %>

<HTML>
<HEAD>
  <LINK rel="stylesheet" type="text/css" href="Main.css">
  <!-- End Style Sheet -->
```

Listing 9.3 continued

```
</HEAD>
<BODY>

<h2>Hyperlink Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <ASP:Hyperlink runat=server
        NavigateUrl="http://www.asppages.com"
        ImageUrl="http://localhost/book/images/test.jpg"
        Target="_NewWindow">
        Click Here to Navigate!
    </ASP:Hyperlink>
</form>

</BODY>
</HTML>
```

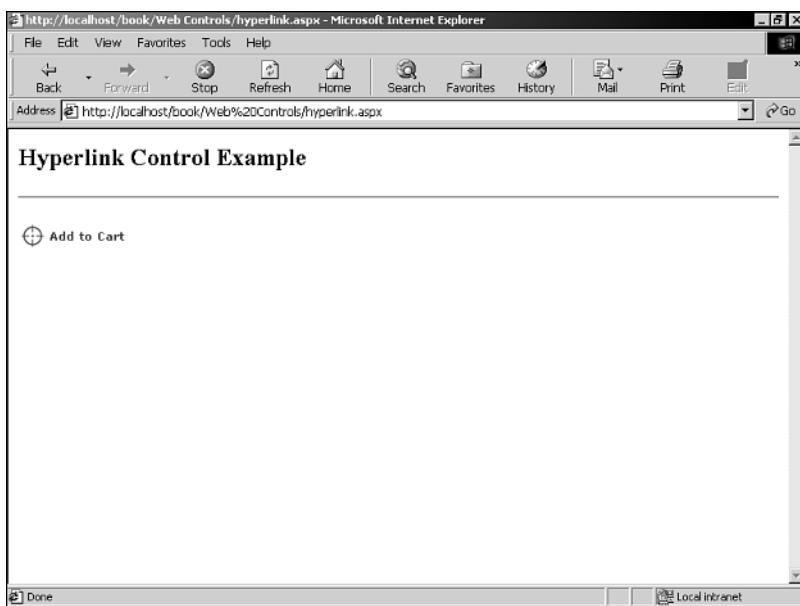


Figure 9.2

A HyperLink Web control.

This example is fairly straightforward. Notice that the `NavigateUrl` attribute defines the `href` attribute of the final HTML tag. `Target` is used to specify the browser window used for navigation.

Also, in this example `Click Here to Navigate!` is placed between the beginning and ending `Hyperlink` tags. It could instead be placed inside the tag itself, as the value of the `Text` attribute.

An attribute that is not used in this example is `ImageUrl`, which allows you to specify the URL of an image that will be used instead of text to create the link. If the `ImageUrl` and `NavigateUrl` attributes are both specified, the `ImageUrl` overrides the `NavigateUrl` attribute.

NOTE

What happens if you set the `href` HTML attribute manually by using GIGO? Most browsers just use the first `href` attribute that they find in order to perform the navigation.

The Image Control

Table 9.3 The Image Control at a Glance

Properties	ImageAlign	ImageAlign
------------	------------	------------

Listing 9.4 An Example of an Image Control (C#)

```
<%@ Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void toggle_image(object source, EventArgs e)
        {
            if(image1.ImageUrl=="/book/images/test1.jpg")
                image1.ImageUrl="/book/images/test2.jpg";
            else
                image1.ImageUrl="/book/images/test1.jpg";
        }
    </script>

</HEAD>
<BODY>

<h2>Image Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:Image id=image1 AlternateText="Image"
```

Listing 9.4 continued

```
ImageUrl="http://localhost/book/images/test1.jpg" runat=server />
<br>
<asp:Button id=button1 Text="Change Image"
    OnClick="toggle_image" runat=server/>
</form>

</BODY>
</HTML>
```

Listing 9.5 An Example of an Image Control (Visual Basic.NET)

```
<%@ Page Language="VB" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >
        Sub toggle_image(sender as Object, e as EventArgs)
            if image1.ImageUrl = "/book/images/test1.jpg" then
                image1.ImageUrl="/book/images/test2.jpg"
            else
                image1.ImageUrl="/book/images/test1.jpg"
            end if
        End Sub
    </script>

</HEAD>
<BODY>

<h2>Image Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:Image id=image1 AlternateText="Image"
        ImageUrl="http://localhost/book/images/test1.jpg" runat=server />
<br>
    <asp:Button id=button1 Text="Change Image"
        OnClick="toggle_image" runat=server/>
</form>

</BODY>
</HTML>
```

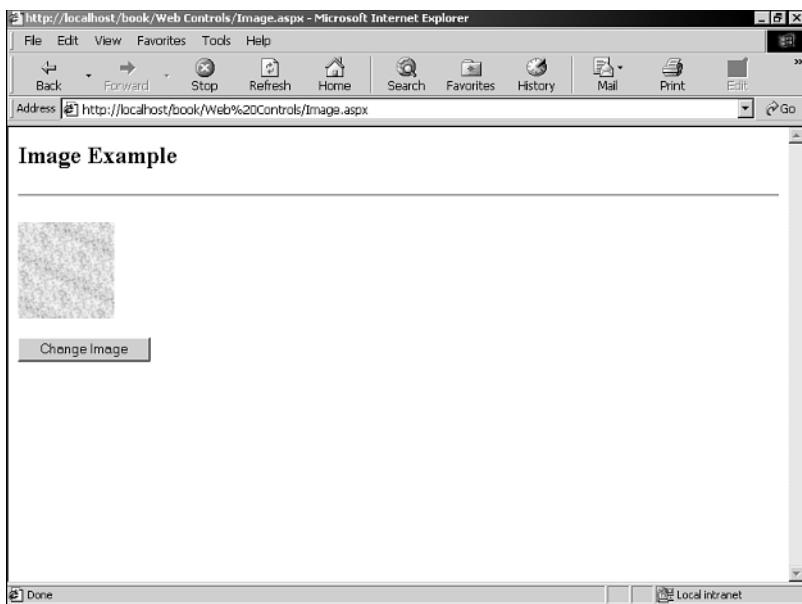


Figure 9.3

An Image Web control.

NOTE

Note the signature of the `ImageButton OnClick()` event. The second argument is of type `ImageClickEventArgs`. If this is specified incorrectly, a compiler error results.

Listings 9.4 and 9.5 display how to programmatically manipulate properties of the Image Web control. `ImageUrl` is the path to the image that is to be displayed. The `AlternateText` attribute is used to enter text that will be seen by clients that have images turned off (including many mobile devices). You can use another attribute, `ImageAlign`, to align an image on a page. `ImageAlign` can contain the following values, as defined in the `System.Web.UI.WebControls.ImageAlign` class: `AbsBottom`, `AbsMiddle`, `Baseline`, `Bottom`, `Left`, `Middle`, `NotSet`, `Right`, `TextTop`, and `Top`.

TIP

It is a good programming practice to use `AlternateText` so that vision-impaired users or users with images turned off can still use your site.

The Panel Control

Table 9.4 The Panel Control at a Glance

Properties	String BackImageUrl
	HorizontalAlign
	Bool wrap

The Panel Web control gives you more control over general page layout. It generates a <div> HTML tag. Listings 9.6 and 9.7 show an example of how to use the Panel control. Figure 9.4 shows how Listings 9.6 and 9.7 appear in Microsoft Internet Explorer 5.5.

Listing 9.6 An Example of a Panel Control (C#)

```
<%@ Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->
    <script language="C#" runat="server">
        void button1_OnClick(Object sender, EventArgs e)
        {
            if(panel1.Enabled==true)
            {
                panel1.Enabled=false;
                button1.Text="Enable Panel";
            }
            else
            {
                panel1.Enabled=true;
                button1.Text="Disable Panel";
            }
        }
    </script>
</HEAD>
<BODY>

<h2>Panel Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:Panel id=panel1 HorizontalAlign="Left"
        title="Sample Panel" tabIndex=1 runat=server>
        I am a panel
        <asp:label Text="Address:" runat=server/>
        <asp:TextBox runat=server/><br>
    </asp:Panel><br><br>
```

Listing 9.6 continued

```
<asp:Button id=button1 Text="Disable Panel"
    OnClick="button1_OnClick" runat=server/>
</form>

</BODY>
</HTML>
```

Listing 9.7 An Example of a Panel Control (Visual Basic.NET)

```
<%@ Page Language="VB" %>

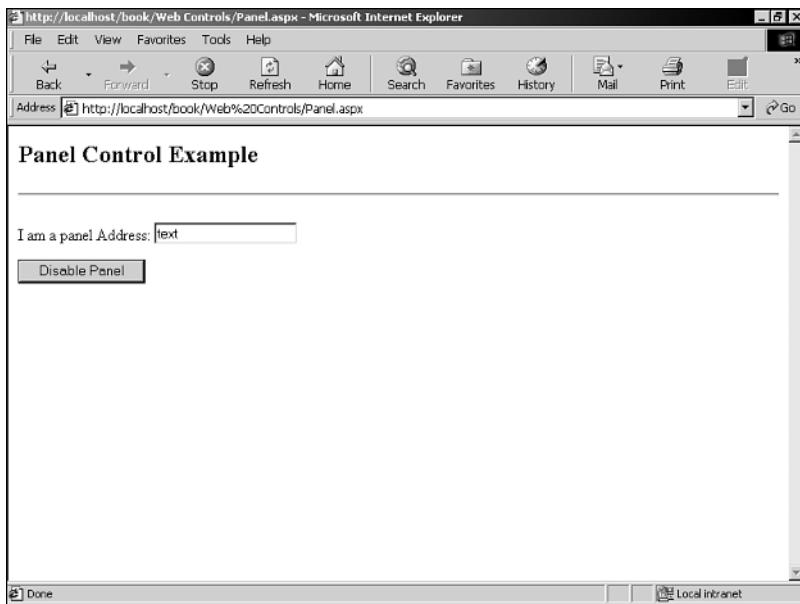
<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->
    <script language="VB" runat="server">
        Sub button1_OnClick(sender as Object, e as EventArgs)
            If panel1.Enabled = True Then
                panel1.Enabled=False
                button1.Text="Enable Panel"
            Else
                panel1.Enabled=True
                button1.Text="Disable Panel"
            End If
        End Sub
    </script>
</HEAD>
<BODY>

<h2>Panel Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:Panel id=panel1 HorizontalAlign="Left"
        title="Sample Panel" tabIndex=1 runat=server>
        I am a panel
        <asp:label Text="Address:" runat=server/>
        <asp:TextBox runat=server/><br>
    </asp:Panel><br><br>

    <asp:Button id=button1 Text="Disable Panel"
        OnClick="button1_OnClick" runat=server/>
</form>

</BODY>
</HTML>
```

**Figure 9.4**

A Panel Web control.

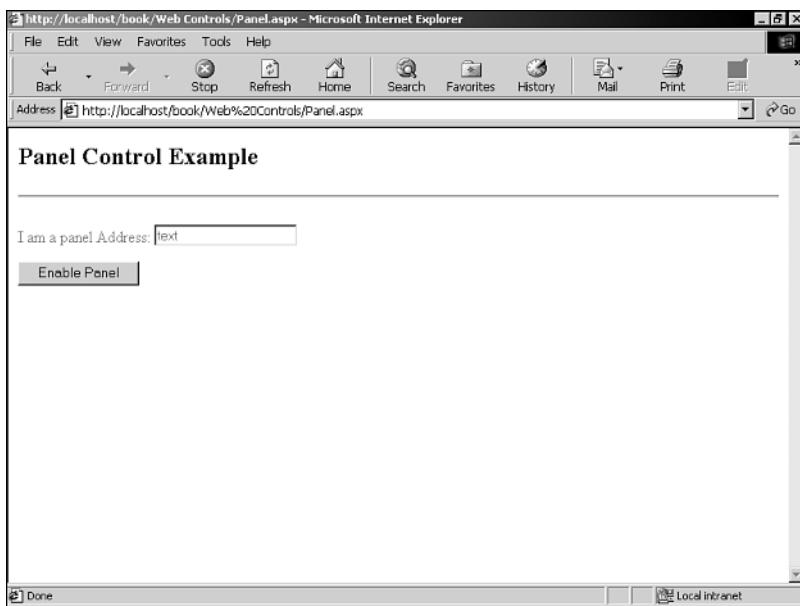
When a panel is disabled, all controls and elements within the panel are also disabled. This makes it handy to group logical sets of Web form elements. Rather than having to write a potentially large function to manually toggle the Enabled state of the controls, you can accomplish this toggling by using one line of code. A disabled panel is grayed out, and users intuitively know that the set of controls are disabled. An example of this is shown in Figure 9.5.

In addition, the BackImageUrl property can be used to specify a background image for the <div> tag. The HorizontalAlign and wrap properties enable you to control the appearance of the <div> tag.

The Table Control

Table 9.5 The Table Control at a Glance

Properties	String BackImageUrl Unit CellSpacing Unit CellPadding GridLines Gridlines HorizontalAlign HorizontalAlign TableRowCollection Rows
------------	--

**Figure 9.5**

A Panel Web control with Disabled set to True.

The Table Web control gives you a consistent way to control the layout of HTML and form elements, using HTML tables. Because the ASP.NET engine automatically handles inconsistencies in the implementation of tables across browsers, you need to be concerned only with the logical table layout. Listings 9.8 and 9.9 show a basic Table control example.

Listing 9.8 An Example of a Table Control (C#)

```
<%@ Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {

            if(IsPostBack==true)
            {
                cell1.Text = "form1 has been submitted.";
            }
        }
    </script>
</HEAD>
<BODY>
    <Form>
        <Table>
            <Tr>
                <Td>
                    <Panel>
                        I am a panel Address:<input type="text" value="text"/>
                        <br/>
                        <input type="button" value="Enable Panel" />
                    </Panel>
                </Td>
            </Tr>
        </Table>
    </Form>
</BODY>
</HTML>
```

Listing 9.8 continued

```
        }
    }

    void Button2_OnClick(Object Source, EventArgs E)
    {
        if( table1.BackImageUrl == "" )
            table1.BackImageUrl="/book/images/test.jpg";
        else
            table1.BackImageUrl="";
    }

</script>

</HEAD>
<BODY>

<h2>Table Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:Table id="table1" Width="100%" CellSpacing="1"
        CellPadding="4" Gridlines="3" runat="server">
        <asp:TableRow>
            <asp:TableCell valign=bottom Wrap="true" ColumnSpan="1" id="cell1">
                <br>Content of cell 1<br>
            </asp:TableCell>
            <asp:TableCell valign=bottom Wrap="true" ColumnSpan="1" id="cell2">
                <br>Content of cell 2<br>
            </asp:TableCell>
            <asp:TableCell valign=bottom ColumnSpan="1" id="cell3">
                <br>Content of cell 3<br>
            </asp:TableCell>
        </asp:TableRow>
    </asp:Table>

    <br>

    <asp:Button Text="Submit" id="button1" runat="server"/>
    <asp:Button Text="Toggle Background Image"
        OnClick="Button2_OnClick" id="button2" runat="server"/>
    <br><br>
</form>

</BODY>
</HTML>
```

Listing 9.9 An Example of a Table Control (Visual Basic.NET)

```
<%@ Page Language="VB" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >
        Sub Page_Load(Source as Object, E as EventArgs)
            if IsPostBack = true then
                cell1.Text = "form1 has been submitted."
            end if
        End Sub

        Sub Button2_OnClick(Source as Object, E as EventArgs)
            if table1.BackImageUrl = "" then
                table1.BackImageUrl="/book/images/test.jpg"
            else
                table1.BackImageUrl=""
            end if
        End Sub

    </script>

</HEAD>
<BODY>

<h2>Table Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:Table id="table1" Width="100%" CellSpacing="1"
        CellPadding="4" Gridlines="3" runat="server">
        <asp:TableRow>
            <asp:TableCell valign=bottom Wrap="true" ColumnSpan="1" id="cell1">
                <br>Content of cell 1<br>
            </asp:TableCell>
            <asp:TableCell valign=bottom Wrap="true" ColumnSpan="1" id="cell2">
                <br>Content of cell 2<br>
            </asp:TableCell>
            <asp:TableCell valign=bottom ColumnSpan="1" id="cell3">
                <br>Content of cell 3<br>
            </asp:TableCell>
        </asp:TableRow>
    </asp:Table>
</form>
```

Listing 9.9 continued

```
<br>
<asp:Button Text="Submit" id="button1" runat="server"/>
<asp:Button Text="Toggle Background Image" OnClick="Button2_
    OnClick" id="button2" runat="server"/><br><br>
</form>

</BODY>
</HTML>
```

The code in Listings 9.8 and 9.9 renders a table to the browser, as in Figure 9.6.

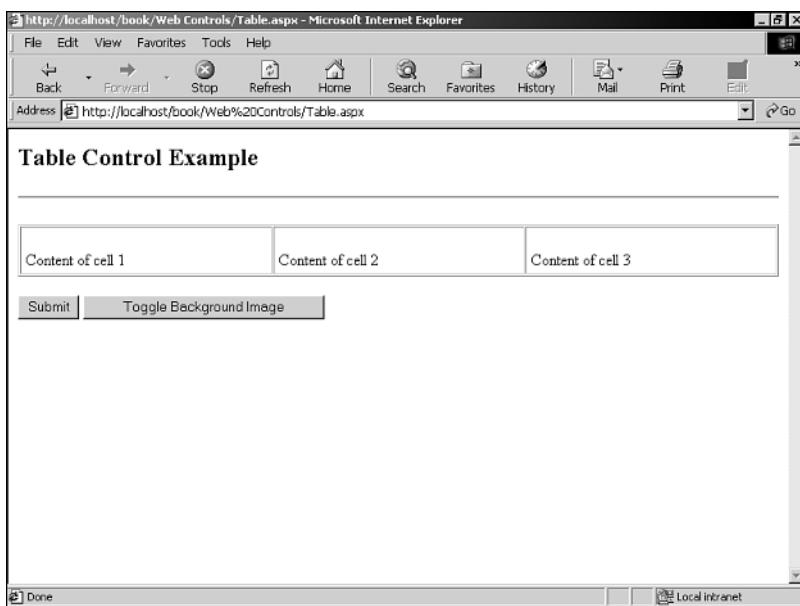


Figure 9.6

A Table Web control.

The Table Web control has several properties that give you control over the generated table's appearance. `BackImageUrl` allows you to specify a background image for the table. In the example shown in Figure 9.6, after a user clicks on the Toggle Background Image button, the table's appearance changes, as shown in Figure 9.7.

The `Gridlines` property allows developers to add borders to the table in the following fashion: A value of 0 specifies no borders. A value of 1 turns borders on for the table and also adds borders for rows only. A value of 2 also turns borders on for the entire table, but it adds borders for columns. Setting `Gridlines` to 3 adds borders for columns, rows, and the table.

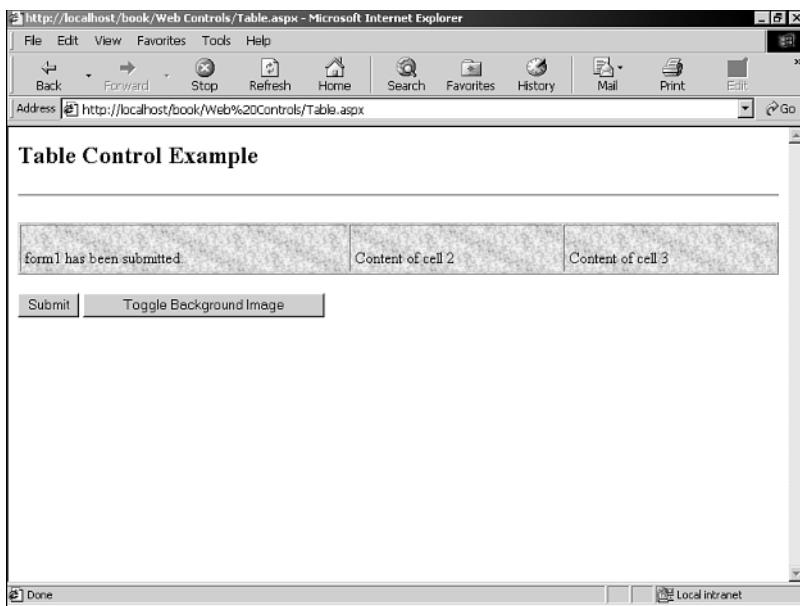


Figure 9.7

A Table Web control with Background Image Toggled.

CellSpacing and CellPadding are precisely the same as their HTML counterparts. CellSpacing is the amount of space between cells in a table. CellPadding denotes the amount of space the cell contents will be kept from the cell borders. You can use the HorizontalAlign property to control the alignment of the table on the page.

The Table Web control is a container object for TableCell and TableRow Web controls, which map directly to standard HTML <td> and <tr> tags, respectively.

TIP

You do not need to explicitly give a Web control an ID. In fact, attempting to maintain uniqueness of all IDs in a Table control can be difficult and is not needed unless you actually need to reference the control (for example, as in Figure 9.7). If no ID is assigned, the page compiler generates one automatically for use in compiling.

Web Controls for Building Forms

The Web controls covered so far in this chapter are non-form related and offer you a consistent object model to use in laying out an ASP.NET page. However, another set of Web controls is especially useful for implementing form elements. For these form-based controls to function properly, they must be placed within a server-side tag, using this format:

```
<form runat="server"></form>
```

The following Web controls are covered in this section:

- **Button**, **LinkButton**, and **ImageButton**—These are used to place various kinds of buttons on the page.
- **CheckBox**—This is used to place a checkbox on the page.
- **CheckBoxList**—This is used to place a grouping of checkboxes on the page.
- **RadioButton**—This is used to place a radio (option) button on the page.
- **TextBox**—This is used to place input textboxes on the page.
- **DropDownList**—This is used to place a drop-down list on the page.



NOTE

Note that only one server-side form can be present on any single ASP.NET page. Adding more than one generates a compiler error.

Button, LinkButton, and ImageButton Controls

Table 9.6 The Button and LinkButton Controls at a Glance

Properties	<code>String Text</code> <code>String BubbleCommand</code> <code>String BubbleArgument</code>
Methods	<code>OnClick(EventArgs e)</code> <code>AddOnClick(EventHandler handler)</code> <code>RemoveOnClick(EventHandler handler)</code>

Table 9.7 The ImageButton Control at a Glance

Properties	<code>int X</code> <code>int Y</code> <code>String BubbleCommand</code> <code>String BubbleArgument</code>
Methods	<code>OnClick(EventArgs e)</code> <code>AddOnClick(EventHandler handler)</code> <code>RemoveOnClick(EventHandler handler)</code>

Three Web controls are used to generate different types of HTML form buttons: **Button**, **LinkButton**, and **ImageButton**. The **Button** Web control is the standard HTML form button that you can find in most forms. The **LinkButton** and **ImageButton** controls offer the same functionality as the **Button** control, except that **LinkButton** is a text link and **ImageButton** is an image link. The **LinkButton** control actually generates a hyperlink that is linked to the Web form through automatically generated client-side script. Listings 9.10 and 9.11 show how these three controls are used.

Listing 9.10 An Example of a Button, LinkButton, and ImageButton Controls (C#)

```
<%@ Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Button1_OnClick(Object Source, EventArgs E)
        {
            msg.Text = "You have clicked button 1.";
        }
        void Button2_OnClick(Object Source, EventArgs E)
        {
            msg.Text = "You have clicked button 2.";
        }
        void Button3_OnClick(Object Source, ImageClickEventArgs E)
        {
            msg.Text = "You have clicked button 3.";
        }
    </script>

</HEAD>
<BODY>

<h2>Button Controls Example</h2>
<hr>

<ASP:Label id="msg" Text="Please click a button." runat="server"/>

<form runat="server" id="form1" name="form1">
    <ASP:Button Text="Standard Button 1"
        OnClick="Button1_OnClick" id="button1" runat="server" />
    <br><br>
    <ASP:LinkButton Text="Link Button 2" OnClick="Button2_OnClick"
        id="button2" runat="server" />
    <br><br>
    <ASP:ImageButton id="button3" src="/book/images/test.jpg"
        OnClick="Button3_OnClick" runat="server" />
</form>

</BODY>
</HTML>
```

Listing 9.11 An Example of a Button, LinkButton, and ImageButton Controls (Visual Basic.NET)

```
<%@ Page Language="VB" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >
        Sub Button1_OnClick(Source as Object, E as EventArgs)
            msg.Text = "You have clicked button 1."
        End Sub
        Sub Button2_OnClick(Source as Object, E as EventArgs)
            msg.Text = "You have clicked button 2."
        End Sub
        Sub Button3_OnClick(Source as Object, E as ImageClickEventArgs)
            msg.Text = "You have clicked button 3."
        End Sub
    </script>

</HEAD>
<BODY>

<h2>Button Controls Example</h2>
<hr>

<ASP:Label id="msg" Text="Please click a button." runat="server"/>

<form runat="server" id="form1" name="form1">
    <ASP:Button Text="Standard Button 1"
        OnClick="Button1_OnClick" id="button1" runat="server"/>
    <br><br>
    <ASP:LinkButton Text="Link Button 2" OnClick="Button2_OnClick"
        id="button2" runat="server"/>
    <br><br>
    <ASP:ImageButton id="button3" src="/book/images/test.jpg"
        OnClick="Button3_OnClick" runat="server"/>
</form>

</BODY>
</HTML>
```

The code in Listings 9.10 and 9.11 generates the Web form shown in Figure 9.8.

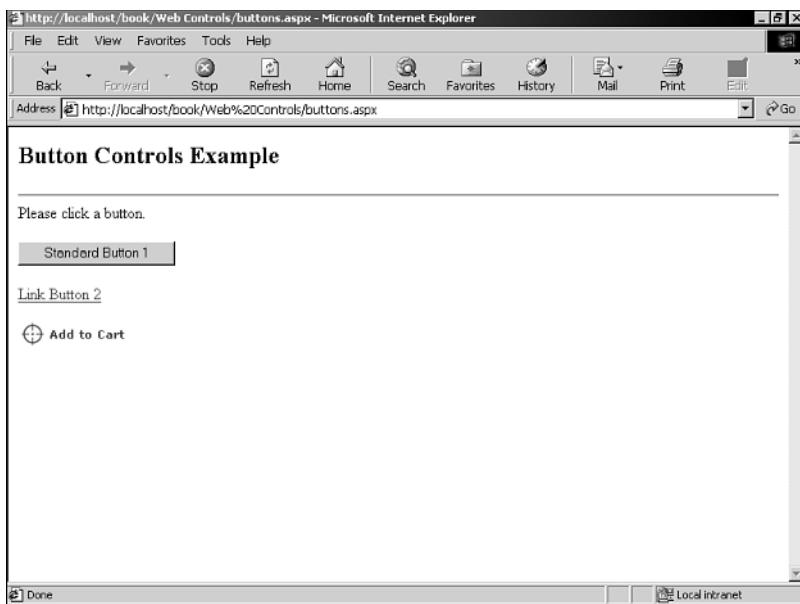


Figure 9.8

The Button, LinkButton, and ImageButton Web controls.

The `Text` property of the `Button` control sets the caption of the generated button (that is, it maps directly to the `value` attribute). The `Text` property of the `LinkButton` control also specifies the text displayed for the generated hyperlink.

All three of these controls have two properties named `BubbleCommand` and `BubbleArgument`:

- `BubbleCommand` is an optional argument that allows you to pass a command automatically to server-side code when the button is clicked. Examples of some commonly used `BubbleCommand` values are `NextPage`, `Sort`, and `Reverse`.
- The `BubbleArgument` property defines an optional command argument that should be passed along with the `BubbleCommand` property when the button is clicked. For example, if the `Sort` command is sent, `ByDate` and `ByName` would make sense as possible arguments.

When a user clicks a button on a page, the form automatically posts to the form processor defined in the `target` property of the form (or to itself, if none is defined). In addition, you can specify a function in the `OnClick` property of each button in order to handle the `OnClick()` event manually.

The CheckBox Control

Table 9.8 The checkBox Control at a Glance

Properties	bool Checked String Text TextAlign TextAlign bool AutoPostBack
Methods	OnCheckedChanged(EventArgs e) AddOnCheckedChanged(EventHandler handler) RemoveOnCheckedChanged(EventHandler handler)

The CheckBox Web control generates a check box HTML element. Check boxes are generally used to present optional choices on a form. (Do not confuse the CheckBox control with the CheckBoxList control that is covered in the next section.) If not specified, the default state for the CheckBox control is unchecked. The CheckBox control generates only a single, standalone check box. Listings 9.12 and 9.13 show a very simple CheckBox control example. Figure 9.9 shows how Listings 9.12 and 9.13 appear in Microsoft Internet Explorer 5.5.

Listing 9.12 An Example of a CheckBox Control (C#)

```
<%@ Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >

        void Button1_OnClick(Object Source, EventArgs E)
        {
            if(checkbox1.Checked==true)
                msg.Text = "Checkbox 1 is checked";
            else
                msg.Text = "Checkbox 1 is not checked";
        }
    </script>

</HEAD>
<BODY>

<h2>CheckBox Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <ASP:Label
        id=msg Text="Please select some items and click the submit button"

```

Listing 9.12 continued

```

    runat=server />
<br><br>
<ASP:CheckBox id="checkbox1" Text="Checkbox 1" runat=server /> <br>
<ASP:Button id="button1" Text="Click Here"
    OnClick="Button1_OnClick" runat=server/>
</form>

</BODY>
</HTML>
```

Listing 9.13 An Example of a CheckBox Control (Visual Basic.NET)

```

<%@ Page Language="VB" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >

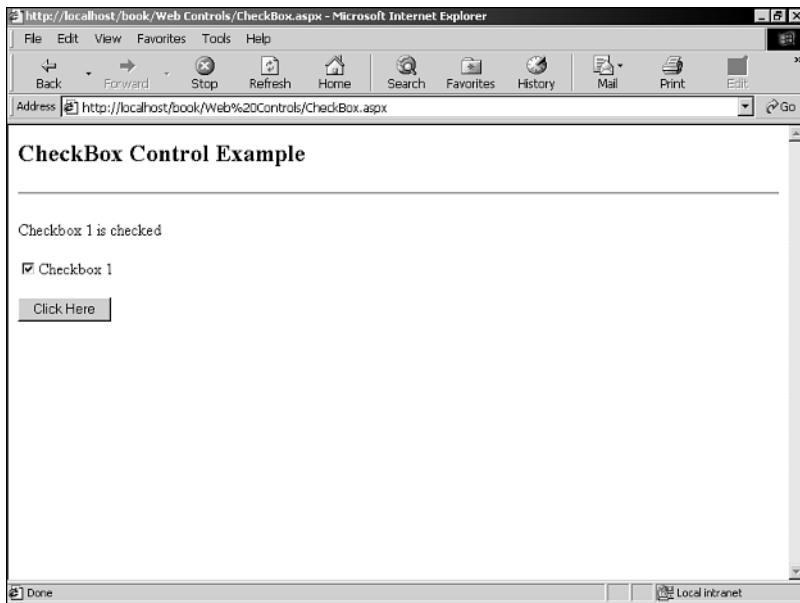
        Sub Button1_OnClick(sender as Object, e as EventArgs)
            if checkbox1.Checked = true then
                msg.Text = "Checkbox 1 is checked"
            else
                msg.Text = "Checkbox 1 is not checked"
            end if
        End Sub
    </script>

</HEAD>
<BODY>

<h2>CheckBox Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <ASP:Label
        id=msg Text="Please select some items and click the submit button" r
        unat=server />
    <br><br>
    <ASP:CheckBox id="checkbox1" Text="Checkbox 1" runat=server /> <br>
    <ASP:Button id="button1" Text="Click Here"
        OnClick="Button1_OnClick" runat=server/>
</form>

</BODY>
</HTML>
```

**Figure 9.9**

A CheckBox Web control.

As shown in Listings 9.12 and 9.13, the `Checked` property can be used to automatically check a check box when a form loads. It can also be used to programmatically detect the state of the control. In addition, the `CheckBox` control has `Text` and `TextAlign` properties that can be used to display and format the text next to the check box. If the `AutoPostBack` property of the `CheckBox` control is set to `True`, then every click will post the form and cause a round-trip back to the server.

The CheckBoxList Control

The `CheckBoxList` Web control is a container item for several check boxes. The developer can use this control to group a set of standard checkboxes together in one logical segment. Listings 9.14 and 9.15 show a simple example of the `CheckBoxList` control. Figure 9.10 shows how Listings 9.14 and 9.15 appear in Microsoft Internet Explorer 5.5.

Listing 9.14 An Example of a CheckBoxList Control (C#)

```
<%@ Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
```

Listing 9.14 continued

```
<!-- End Style Sheet -->

<script language="C#" runat="server" >

    string output;

    void button1_Click(object Source, EventArgs e) {

        msg.Text = "The following checkboxes are checked:<br>";

        for (int i=0; i < check1.Items.Count; i++)
        {
            if ( check1.Items[ i ].Selected )
            {
                // List the selected items
                output = output + check1.Items[i].Text + "\n<br>";
            }
        }

        msg.Text = msg.Text + output;
    }
</script>

</HEAD>
<BODY>

<h2>CheckBoxList Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:CheckBoxList id=check1 runat="server">
        <asp:ListItem>ListItem 1</asp:ListItem>
        <asp:ListItem>ListItem 2</asp:ListItem>
        <asp:ListItem>ListItem 3</asp:ListItem>
    </asp:CheckBoxList><br><br>
    <asp:Button id=button1 Text="Submit"
        onclick="button1_Click" runat="server"/>
    <br><br>
    <asp:Label id=msg Text="Check some items to the left"
    ↪runat="server"/><br>
</form>

</BODY>
</HTML>
```

Listing 9.15 An Example of a CheckBoxList Control (Visual Basic.NET)

```
<%@ Page Language="VB" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >

        dim output as string

        Sub button1_Click(sender as Object, e as EventArgs)
            dim i as integer
            for i = 0 to check1.Items.Count - 1
                if check1.Items(i).Selected then
                    ' List the selected items
                    output = output & check1.Items(i).Text & "<br>"
                end if
            next i

            msg.Text = "The following checkboxes are checked:<br>" & output
        End Sub

    </script>

</HEAD>
<BODY>

<h2>CheckBoxList Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:CheckBoxList id=check1 runat="server">
        <asp:ListItem>ListItem 1</asp:ListItem>
        <asp:ListItem>ListItem 2</asp:ListItem>
        <asp:ListItem>ListItem 3</asp:ListItem>
    </asp:CheckBoxList><br><br>
    <asp:Button id=button1 Text="Submit"
        onclick="button1_Click" runat="server"/>
    <br><br>
    <asp:Label id=msg Text="Check some items above" runat="server"/><br>
</form>

</BODY>
</HTML>
```

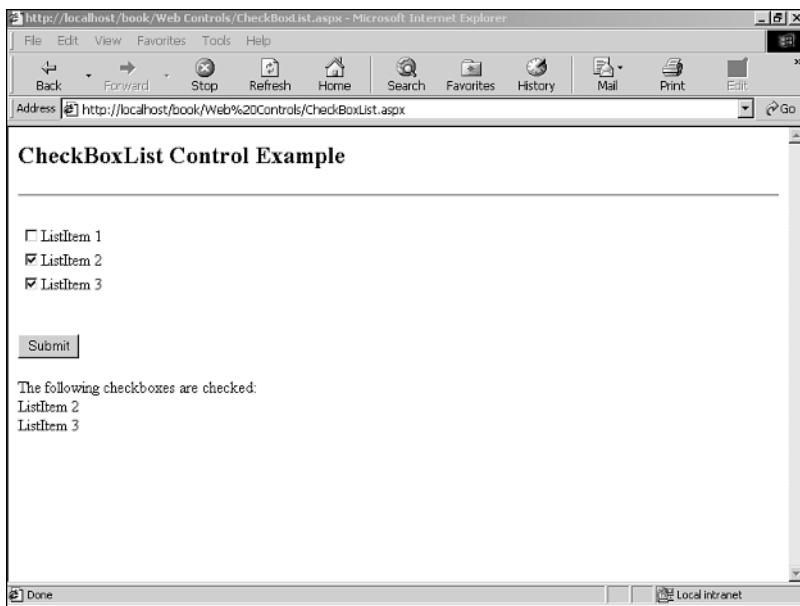


Figure 9.10

A CheckBoxList Web control.

As shown in Listings 9.14 and 9.15, the CheckBoxList Web control is placed on the page by using the standard Web control syntax. However, each individual check box is actually placed on the page inside the CheckBoxList tag by using the ListItem Web control. The individual check boxes are then accessed programmatically through the Items collection of the CheckBoxList control itself, as in Listings 9.14 and 9.15.

The RadioButton Control

Table 9.9 The RadioButton Control at a Glance

Properties	<code>bool Checked</code> <code>String GroupName</code> <code>String Text</code> <code>TextAlign TextAlign</code> <code>bool AutoPostBack</code>
Methods	<code>OnCheckedChanged(EventArgs e)</code> <code>AddOnCheckedChanged(EventHandler handler)</code> <code>RemoveOnCheckedChanged(EventHandler handler)</code>

The RadioButton Web control allows you to place a series of radio buttons (or option button) on a page. Radio buttons function much like check boxes do, except that only one radio button in the group can be chosen at any one time. If none of the radio buttons are set to True, the first radio button is set to True at runtime. Listings 9.16 and

9.17 show how to use the RadioButton control. Figure 9.11 shows how Listings 9.16 and 9.17 appear in Microsoft Internet Explorer 5.5.

Listing 9.16 An Example of a RadioButton Control (C#)

```
<%@ Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {
            if(IsPostBack==true)
            {
                if (radio1.Checked) {
                    msg.Text = "You selected " + radio1.Text;
                }
                else if (radio2.Checked) {
                    msg.Text = "You selected " + radio2.Text;
                }
                else if (radio3.Checked) {
                    msg.Text = "You selected " + radio3.Text;
                }
            }
        }
    </script>

</HEAD>
<BODY>

<h2>RadioButton Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <ASP:RadioButton id=radio1 AutoPostBack="true" Text="Option 1"
        Checked="true" GroupName="bullets1" runat=server /><br>
    <ASP:RadioButton id=radio2 AutoPostBack="true" Text="Option 2"
        Checked="false" GroupName="bullets1" runat=server /><br>
    <ASP:RadioButton id=radio3 AutoPostBack="true" Text="Option 3"
        Checked="false" GroupName="bullets1" runat=server /><br>
    <br>
    <asp:Label id=msg Text="Please make a selection" runat=server/>
</form>

</BODY>
</HTML>
```

Listing 9.17 An Example of a RadioButton Control (Visual Basic.NET)

```
<%@ Page Language="VB" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

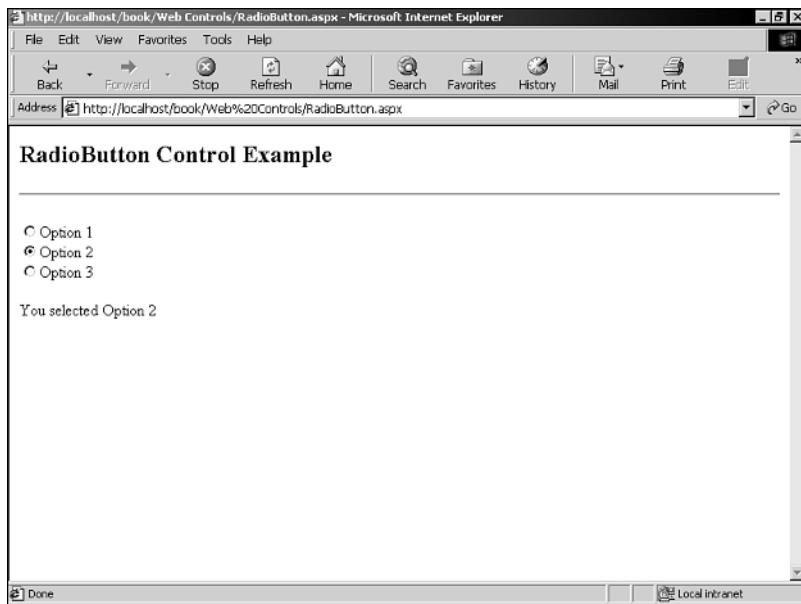
    <script language="VB" runat="server" >
        Sub Page_Load(Source as Object, E as EventArgs)
            If IsPostBack = true Then
                If radio1.Checked Then
                    msg.Text = "You selected " + radio1.Text
                Else If (radio2.Checked) Then
                    msg.Text = "You selected " + radio2.Text
                Else If (radio3.Checked) Then
                    msg.Text = "You selected " + radio3.Text
                End If
            End If
        End Sub
    </script>

</HEAD>
<BODY>

<h2>RadioButton Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <ASP:RadioButton id=radio1 AutoPostBack="true" Text="Option 1"
        Checked="true" GroupName="bullets1" runat=server /><br>
    <ASP:RadioButton id=radio2 AutoPostBack="true" Text="Option 2"
        Checked="false" GroupName="bullets1" runat=server /><br>
    <ASP:RadioButton id=radio3 AutoPostBack="true" Text="Option 3"
        Checked="false" GroupName="bullets1" runat=server /><br>
    <br>
    <asp:Label id=msg Text="Please make a selection" runat=server/>
</form>

</BODY>
</HTML>
```

**Figure 9.11**

A RadioButton Web control.

You group RadioButton controls are grouped on a page by using the `GroupName` property. Only one radio button in each group can be selected at one time. The standard `Text` and `TextAlign` properties enable you to place and arrange text next to a radio button. The `Checked` property allows you to programmatically check whether a given radio button is selected and also allows you to set it. Listings 9.16 and 9.17 are implemented without a button, by using the `AutoPostBack` property of the RadioButton control.

The TextBox Control

Table 9.10 The TextBox Control at a Glance

Properties	<code>int Columns</code> <code>int Rows</code> <code>int MaxLength</code> <code>String Text</code> <code>TextBoxMode TextMode</code> <code>bool Wrap</code> <code>bool AutoPostBack</code>
Methods	<code>OnTextChanged(EventArgs e)</code> <code>AddOnTextChanged(EventHandler handler)</code> <code>RemoveOnTextChanged(EventHandler handler)</code>

The TextBox Web control generates an HTML textbox. This is a very versatile control that gathers user keyboard input. The TextBox Web control has three separate modes

(as defined by the `TextBoxMode` class). If no mode is specified, the default is `SingleLine`. This creates a single-line text input box. If `SingleLine` is specified, then the `Wrap` and `Rows` properties are unavailable. If `TextMode` is set to `Password`, then a password input box is generated. The password input box is a single-line input box, and when characters are typed into the input box, they are echoed back to the screen as asterisks for security purposes. If `MultiLine` is specified, a multiline text box is generated, and the `Wrap` and `Rows` properties are available. Listings 9.18 and 9.19 display all three uses of the `TextBox` control. Figure 9.12 shows how Listings 9.18 and 9.19 appear in Microsoft Internet Explorer 5.5.

Listing 9.18 An Example of a TextBox Control (C#)

```
<%@ Page Language="C#" %>

<html>
<head>
    <link rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->
</head>
<body>

<h2>TextBox Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <table>
        <tr><td>
            Name: </td>
        <td>
            <asp:TextBox id="name" MaxLength="10" Columns="30" runat=server/></td></tr>
        <tr><td>
            Password:
        </td>
        <td>
            <asp:TextBox id="password" MaxLength="10"
                TextMode="Password" Columns="30" runat=server/></td></tr>
        <tr><td>
            Notes:
        </td>
        <td>
            <asp:TextBox id="notes" TextMode="MultiLine" Wrap="true"
                MaxLength="10" Columns="50" Rows="5" runat=server/></td></tr>
    </table>
</form>

</body>
</html>
```

Listing 9.19 An Example of a TextBox Control (Visual Basic.NET)

```
<%@ Page Language="C#" %>

<html>
<head>
    <link rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->
</head>
<body>

<h2>TextBox Example</h2>
<hr>

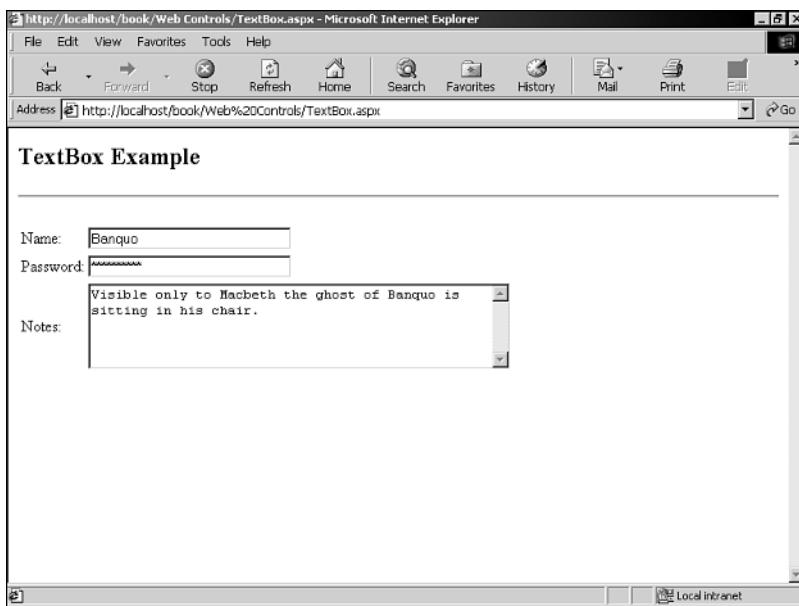
<form runat="server" id=form1 name=form1>
    <table>
        <tr><td>
            Name: </td>
        <td>
            <asp:TextBox id="name" MaxLength="10" Columns="30" runat=server/></td></tr>
        <tr><td>
            Password:
        </td>
        <td>
            <asp:TextBox id="password" MaxLength="10" TextMode="Password"
                Columns="30" runat=server/></td></tr>
        <tr><td>
            Notes:
        </td>
        <td>
            <asp:TextBox id="notes" TextMode="MultiLine" Wrap="true"
                MaxLength="10" Columns="50" Rows="5" runat=server/></td></tr>
    </table>
</form>

</body>
</html>
```

The horizontal width of the text box can be set on a character-by-character basis, using the `Columns` property. The `MaxLength` property maps directly to the `maxlength` HTML attribute, which allows you to limit the text box to a certain number of characters to prevent buffer overrun attacks or to prevent a user from entering more data than there is room for in a database field.

NOTE

The `TextBox` control has an `AutoPostBack` property, but it is unclear why anyone would use it. If `AutoPostBack` is set to `True`, every keystroke a user enters into the text box will cause the form to be posted, costing a round-trip to the server.

**Figure 9.12**

A *TextBox* *Web control*.

The DropDownList Control

Table 9.11 The DropDownList Control at a Glance

Properties	<code>ICollection DataSource</code> <code>String DataFieldText</code> <code>String DataFieldValue</code> <code>ListItemCollection Items</code> <code>int SelectedIndex</code> <code>ListItem SelectedItem</code> <code>Bool AutoPostBack</code>
Methods	<code>AddOnSelectedIndexChanged(EventHandler value)</code> <code>RemoveOnSelectedIndexChanged(EventHandler value)</code> <code>OnSelectedIndexChanged(EventArgs e)</code>

The *DropDownList* *Web control* generates a `<select>` *HTML tag*. You use the *DropDownList* control when you want to give the user a number of different choices, of which only one can be chosen. Listings 9.20 and 9.21 show how to add a *DropDownList* control to a form. Figure 9.13 shows how Listings 9.20 and 9.21 appear in Microsoft Internet Explorer 5.5.

Listing 9.20 An Example of a DropDownList Control (C#)

```
<%@ Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {
            if(IsPostBack==true)
            {
                msg.Text = "You have selected '" 
                    + dropdownlist1.SelectedItem.Text + "'";
            }
        }
    </script>

</HEAD> HTML tag
<BODY>

<h2>DropDownList Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:dropdownlist id=dropdownlist1 AutoPostBack=true runat=server>
        <asp:Listitem>ListItem One</asp:Listitem>
        <asp:Listitem>ListItem Two</asp:Listitem>
        <asp:Listitem>ListItem Three</asp:Listitem>
        <asp:Listitem>ListItem Four</asp:Listitem>
    </asp:dropdownlist>
    <br><br>
    <asp:Label Text="Please make a selection" id=msg runat="server" />
</form>

</BODY>
</HTML>
```

Listing 9.21 An Example of a DropDownList Control (Visual Basic.NET)

```
<%@ Page Language="VB" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >
```

Listing 9.21 continued

```

Sub Page_Load(sender as Object, e as EventArgs)
    if IsPostBack = true then
        msg.Text = "You have selected '" +
            + dropdownlist1.SelectedItem.Text + "'"
    end if
End Sub
</script>

</HEAD>
<BODY>

<h2>DropDownList Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:dropdownlist id=dropdownlist1 AutoPostBack=true runat=server>
        <asp:Listitem>ListItem One</asp:Listitem>
        <asp:Listitem>ListItem Two</asp:Listitem>
        <asp:Listitem>ListItem Three</asp:Listitem> HTML tag
        <asp:Listitem>ListItem Four</asp:Listitem>
    </asp:dropdownlist>
    <br><br>
    <asp:Label Text="Please make a selection" id=msg runat="server" />
</form>

</BODY>
</HTML>

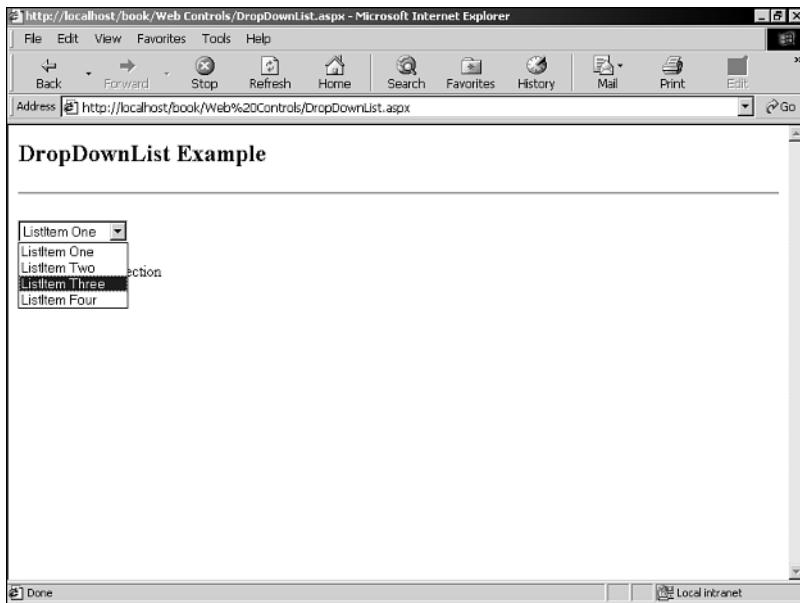
```

As you can see in Figure 9.13, the `DropDownList` control is a container for list items—that is, you can add individual elements of the drop-down list by using the `<asp:Listitem>` tag. It is possible to use the `Items` collection of the `DropDownList` control to iterate through all possible items in the list. You use the `SelectedItem` property of the `DropDownList` control to find the item that the user has chosen. You can use the `SelectedIndex` property in the same manner, returning the integer index of the selected item; you can also use it to set the selected item (for instance, in order to bring up the user's previous choices).

The list items have `Text`, `Value`, and `Selected` properties. You use `Selected` to make a value the default when the control is instantiated, and you use `Text` to set the text the user will see for the item. `Value` is the underlying value for a list item.

The Data-Bound DropDownList Control

In addition to manually defining the list items, you can bind the `DropDownList` to a data source. Listings 9.22 and 9.23 show how this is done. Figure 9.14 shows how Listings 9.22 and 9.23 appear in Microsoft Internet Explorer 5.5.

**Figure 9.13**

A DropDownList Web control.

Listing 9.22 An Example of a Data-Bound DropDownList Control (C#)

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.OleDb" %>

<HTML>
<HEAD>
    <!-- Insert Style Sheet Here -->
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {
            if(IsPostBack==false)
            {
                string sConn = "server=localhost;Database=Northwind;User
                    ID=Sa;Password=;Provider=SQLOLEDB";
                string sSQL = "Select FirstName, EmployeeID FROM Employees";

                OleDbConnection cn = new OleDbConnection(sConn);
            }
        }
    </script>
</HEAD>
<body>
    <form>
        <table border="1">
            <tr>
                <td>First Name:</td>
                <td><asp:DropDownList ID="DropDownList1" runat="server" /></td>
            </tr>
            <tr>
                <td>Last Name:</td>
                <td><asp:TextBox ID="TextBox1" runat="server" /></td>
            </tr>
        </table>
    </form>
</body>

```

Listing 9.22 continued

```
OleDbDataAdapter adapter = new OleDbDataAdapter(sSQL, cn);

DataSet ds = new DataSet();

adapter.Fill(ds, "Employees");

OrderList.DataSource = ds.Tables[0].DefaultView;
OrderList.DataBind();

lblMessage.Text = "";
lblMessage2.Text = "";

OrderList.SelectedIndex = 3;

}

}

void btnOS_click(Object Sender, EventArgs E)
{
    lblMessage.Text = "You Selected: " + OrderList.SelectedItem.Text;
    lblMessage2.Text = "Selected Index = " + OrderList.SelectedIndex;

}

</script>

</HEAD>
<BODY>

<h2>DropDownList Control With DataBinding</h2>
<hr>

<form id=form1 name=form1 runat="server">

    <asp:DropDownList id="OrderList" runat="server"
        DataTextField="Fullname" DataValueField="CustomerID"/>

    <asp:Button id="btnOS" onclick="btnOS_click"
        Text="Select" runat="server"/><br>
    <asp:Label id="lblMessage" runat="server" />
    <asp:label id="lblMessage2" runat="server"></asp:label>

</form>

</BODY>
</HTML>
```

Listing 9.23 An Example of a Data-Bound DropDownList Control (Visual Basic.NET)

```
<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.OleDb" %>

<HTML>
<HEAD>
    <!-- Insert Style Sheet Here -->
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

<script language="VB" runat="server" >
    Sub Page_Load(sender as Object, e as EventArgs)
        If IsPostBack = false Then
            Dim sConn, sSQL As String
            Dim cn As OleDbConnection
            Dim ds As DataSet
            Dim adapter As OleDbDataAdapter

            sConn = "server=localhost;Database=Northwind;User _"
            ID=Sa;Password=;Provider=SQLOLEDB;"
            sSQL = "Select EmployeeID, FirstName AS Name FROM Employees"

            cn = New OleDbConnection(sConn)
            adapter = New OleDbDataAdapter(sSQL, cn)

            ds = New DataSet()
            adapter.Fill(ds, "Employees")

            OrderList.DataSource = ds.Tables(0).DefaultView
            OrderList.DataBind()

            lblMessage.Text = ""
            lblMessage2.Text = ""

            OrderList.SelectedIndex = 3
        End If
    End Sub

    Sub btnOS_click(sender as Object, e as EventArgs)
        lblMessage.Text = "You Selected: " & OrderList.SelectedItem.Text
        lblMessage2.Text = "Selected Index = " & OrderList.SelectedIndex
    End Sub
</script>

</HEAD>
<BODY>
```

Listing 9.23 continued

```
<h2>DropDownList Control With DataBinding</h2>
<hr>

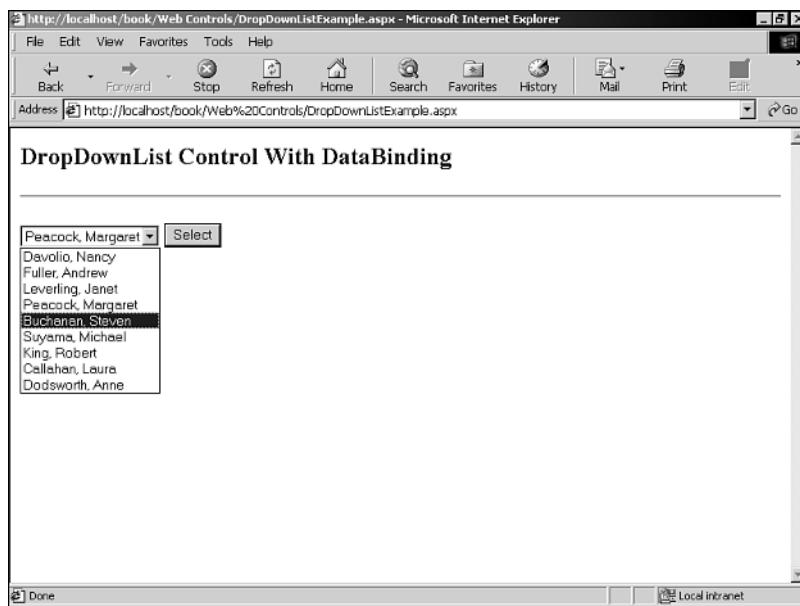
<form id=form1 name=form1 runat="server">

    <asp:DropDownList id="OrderList" runat="server"
        DataTextField="Name" DataValueField="EmployeeID"/>

    <asp:Button id="btnOS" onclick="btnOS_click"
        Text="Select" runat="server"/><br>
    <asp:Label id="lblMessage" runat="server" />
    <asp:label id="lblMessage2" runat="server"></asp:label>

</form>

</BODY>
</HTML>
```

**Figure 9.14**

A *Data-Bound* DropDownList Web control.

Listings 9.22 and 9.23 use data binding. Since this is the first time you might have seen this, here is a brief explanation of the process:

1. A connection object (`cn`) is created to serve as the connection to the database.
2. A dataset command is created, which in this case returns a list of authors from the Pubs default database in Microsoft SQL 7.0.
3. The dataset itself is instantiated and filled with the employee data from the database.
4. The `DataSource` collection of the `DropDownList` control is used to specify the data source used to fill the drop-down list.
5. The `DataBind()` method binds the control to the data set.

For more information on data binding List controls, please see Chapter 11, “Working with List Controls and Data Binding.”

The ListBox Control

Table 9.12 The ListBox Control at a Glance

Properties	<code>IList DataSource</code> <code>String DataFieldText</code> <code>String DataFieldValue</code> <code>ListItemCollection Items</code> <code>int Rows</code> <code>int SelectedIndex</code> <code>ListItem SelectedItem</code> <code>ListBoxSelectionMode SelectionMode</code> <code>bool AutoPostBack</code>
Methods	<code>AddOnSelectedIndexChanged(EventArgs value)</code> <code>RemoveOnSelectedIndexChanged(EventArgs value)</code> <code>OnSelectedIndexChanged(EventArgs e)</code>

The `ListBox` Web control is used to display multiple items in a box. Just like the `DropDownList` control, the `ListBox` control generates a `<select>` HTML tag. Even though `Size` is not an explicit property of the `ListBox` control, if it is set to 1 in the `ListBox` tag and then passed (using GIGO) to the page, the list box looks just like a drop-down list. Listings 9.24 and 9.25 show a practical example of a `ListBox` control. Figure 9.15 shows how Listings 9.24 and 9.25 appear in Microsoft Internet Explorer 5.5.

Listing 9.24 An Example of a ListBox Control (C#)

```
<%@ Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Button1_OnClick(Object Source, EventArgs E)
    {
```

Listing 9.24 continued

```
if(listbox1.SelectedIndex!=-1)
{
    listbox2.Items.Add(listbox1.SelectedItem.Text);
    listbox1.Items.Remove(listbox1.SelectedItem);
}
}

void Button2_OnClick(Object Source, EventArgs E)
{
    if(listbox2.SelectedIndex!=-1)
    {
        listbox1.Items.Add(listbox2.SelectedItem.Text);
        listbox2.Items.Remove(listbox2.SelectedItem);
    }
}
</script>

</HEAD>
<BODY>

<h2>ListBox Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
<table><tr>
<td>
    <asp:listbox id="listbox1" VisibleItems=5 Style="Width:100" runat=server>
        <asp:Listitem>One</asp:Listitem>
        <asp:Listitem>Two</asp:Listitem>
        <asp:Listitem>Three</asp:Listitem>
    </asp:listbox></td>
<td>
    <asp:Button id="button2" OnClick="Button2_OnClick" Text="<" runat=server/>
    <asp:Button id="button1" OnClick="Button1_OnClick" Text=">" runat=server/>
</td>
<td>
    <asp:listbox id="listbox2" VisibleItems=5 Style="Width:100" runat=server>
    </asp:listbox></td>
</tr></table>
<br>

</form>

</BODY>
</HTML>
```

Listing 9.25 An Example of a ListBox Control (Visual Basic.NET)

```
<%@ Page Language="VB" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >
        Sub Button1_OnClick(sender as Object, e as EventArgs)
            if listbox1.SelectedIndex <> -1 then
                listbox2.Items.Add(listbox1.SelectedItem.Text)
                listbox1.Items.Remove(listbox1.SelectedItem)
            end if
        End Sub

        Sub Button2_OnClick(sender as Object, e as EventArgs)
            if listbox2.SelectedIndex <> -1 then
                listbox1.Items.Add(listbox2.SelectedItem.Text)
                listbox2.Items.Remove(listbox2.SelectedItem)
            end if
        End Sub
    </script>

</HEAD>
<BODY>

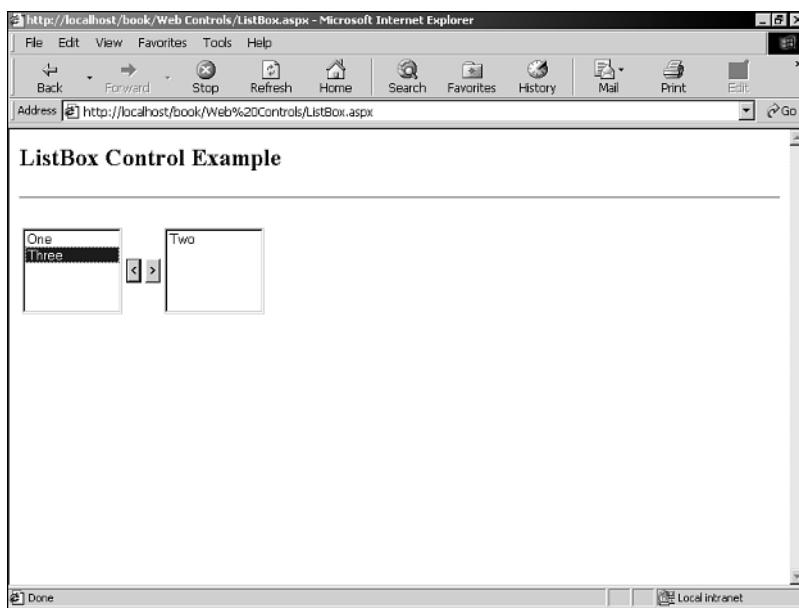
<h2>ListBox Control Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <table><tr>
        <td>
            <asp:listbox id="listbox1" VisibleItems=5 Style="Width:100" runat=server>
                <asp:Listitem>One</asp:Listitem>
                <asp:Listitem>Two</asp:Listitem>
                <asp:Listitem>Three</asp:Listitem>
            </asp:listbox></td>
        <td>
            <asp:Button id="button2" OnClick="Button2_OnClick" Text="<" runat=server/>
            <asp:Button id="button1" OnClick="Button1_OnClick" Text=">" runat=server/>
        </td>
        <td>
            <asp:listbox id="listbox2" VisibleItems=5 Style="Width:100" runat=server>
            </asp:listbox></td>
        </tr></table>
        <br>
```

Listing 9.25 continued

```
</form>

</BODY>
</HTML>
```

**Figure 9.15**

A *ListBox* Web control.

Just like the *DropDownList* control, the *ListBox* control is a container for list items that are added by using the `<asp:ListItem>` tag. You can set the *SelectionMode* property to either *Single* or *Multiple*. If *SelectionMode* is set to *Single*, the user can choose only one list item at any time; if *SelectionMode* is set to *Multiple*, the user can choose more than one list item.

You use the *SelectedItem* and *SelectedIndex* properties to get the value and the index of the selected item, respectively. You use the *Rows* property to set the number of rows the list box displays. As with most other HTML form-based Web controls, the *ListBox* control has an *AutoPostBack* property. If *AutoPostBack* is set to *True*, then each click on a list item will force a postback to occur, and a round-trip to the server will be made.

NOTE

The *ListBox* control can be bound to data in precisely the same manner as the *DropDownList* control.

CHAPTER 10

Encapsulating ASP.NET Page Functionality with User Controls

In previous versions of ASP, if you needed to encapsulate a block of HTML or ASP, you had few choices. The easiest and most common solution was to put commonly used sections of code in a text file and then pull that code into the page by using `include` statements wherever needed.

ASP.NET user controls extend the concept of including files. They not only encapsulate blocks of HTML, but also give you an object-oriented way of encapsulating site functionality. They do this through the use of public and private methods and properties.

Creating a Simple User Control

This section walks you through the creation of a very simple user control that encapsulates the functionality of a typical Web site menu header. The header is encapsulated into a single file and can then be instantiated on any page in the site by using just a few lines of code.

The first step in creating a user control is to create in your ASP.NET application directory a file that has the extension `.ascx`. This file will house the contents of the user control. Just about any kind of text can be placed in the file. For the purposes of this example, Listing 10.1 shows the contents of `Simple.ascx`.

Listing 10.1 A Simple User Control

```
<!--  
This user control is designed to simply output  
a page header to be used on every page.  
-->  
  
<h2>Welcome to XYZ Widgets, Inc!</h2>  
<hr>  
<table>  
  <tr>  
    <td><a href="">Home</a> |</td>  
    <td><a href="">Login</a> |</td>  
    <td><a href="">Latest Updates</a> |</td>  
    <td><a href="">Downloads</a> |</td>  
    <td><a href="">Technical Support</a> |</td>  
    <td><a href="">Help</a></td>  
  </tr>  
</table>  
<hr>
```

NOTE

You can create user controls by using a different Microsoft .NET framework programming language than is used in the main page. For instance, the main page can be created in C# and the user control can be created in Visual Basic.NET.

Registering a User Control on a Page

Before you can use the newly created `Simple.ascx` user control, the control must first be registered on the page where it will be placed. You do this by using the `Register` page directive, as shown in the following line of code:

```
<%@ Register TagPrefix="PureASP" TagName="Simple" Src="Simple.ascx"%>
```

The `TagPrefix` attribute exists primarily to avoid naming conflicts between similarly named user controls. Typically, the `TagPrefix` attribute generally contains a company name or an application name. The `TagName` attribute is the name of the tag used to instantiate the user control on the page. Finally, the `Src` attribute is used to specify the location of the file that encapsulates the user control.

Creating an Instance of a User Control

When a control is registered with the page, instantiating the control is simple and much like instantiating any of the Web controls that are built in to ASP.NET. To create an instance of the `Simple.ascx` user control, you add the following line anywhere on the calling page:

```
<PureASP:Simple runat="server"></PureASP:Simple>
```

The text placed in the `Simple.ascx` file is inserted inline at any point in the page where the control is instantiated. For example, Listing 10.2 shows a complete ASP.NET page that uses the `Simple.ascx` user control.

Listing 10.2 Instantiating a User Control

```
<% @Page Language="C#" %>
<%@ Register TagPrefix="PureASP" TagName="Simple" Src="Simple.ascx"%>

<HTML>
<BODY>

<!--Header-->
<PureASP:Simple runat="server"></PureASP:Simple>

<!--Page Content-->
Today's featured widget is a testament to Aiur...

<form runat="server" id=form1 name=form1>
</form>

</BODY>
</HTML>
```

The output of this page is shown in Figure 10.1.

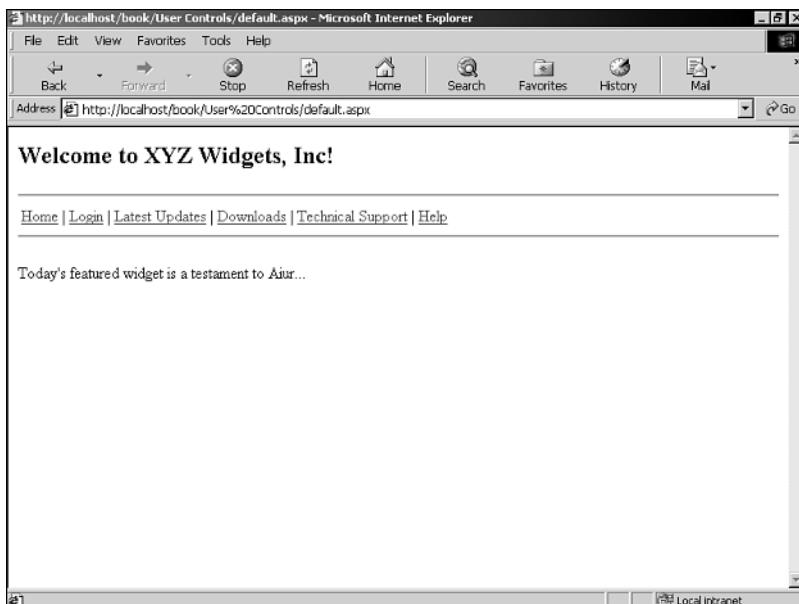


Figure 10.1

Using a user control to encapsulate a site menu.

Adding Properties to a User Control

In addition to simply encapsulating blocks of HTML, user controls can also encapsulate sets of properties and methods. There are two ways to expose public properties in a user control. The first and easiest way is to simply declare a public variable in the user control. Then, the calling page can access the variable by using `myUserControl1.PropertyName` syntax. For example, you can add a `userName` public variable and an ASP label Web control to the `Simple.ascx` user control as shown in Listings 10.3 and 10.4.

Listing 10.3 Adding a Public Property to a User Control (C#)

```
<script language="C#" runat="server" >

    public string userName;

    void Page_Load(Object Source, EventArgs E)
    {
        if( userName != "" )
            msg.Text = "Welcome, " + userName + "!";
    }
</script>

<!--
This user control is designed to simply output
a page header to be used on every page.
-->

<h2>Welcome to XYZ Widgets, Inc!</h2>
<hr>
<table>
    <tr>
        <td><a href="">Home</a> |</td>
        <td><a href="">Login</a> |</td>
        <td><a href="">Latest Updates</a> |</td>
        <td><a href="">Downloads</a> |</td>
        <td><a href="">Technical Support</a> |</td>
        <td><a href="">Help</a></td>
    </tr>
</table>
<hr>
<asp:Label id=msg runat="server"></asp:Label><br>
```

Listing 10.4 Adding a Public Property to a User Control (Visual Basic.NET)

```
<script language="VB" runat="server" >

    public string userName;
```

Listing 10.4 continued

```

void Page_Load(Object Source, EventArgs E)
{
    if( userName != "" )
        msg.Text = "Welcome, " + userName + "!";
}
</script>

<!--
This user control is designed to simply output
a page header to be used on every page.
-->

<h2>Welcome to XYZ Widgets, Inc!</h2>
<hr>
<table>
    <tr>
        <td><a href="">Home</a> |</td>
        <td><a href="">Login</a> |</td>
        <td><a href="">Latest Updates</a> |</td>
        <td><a href="">Downloads</a> |</td>
        <td><a href="">Technical Support</a> |</td>
        <td><a href="">Help</a></td>
    </tr>
</table>
<hr>
<asp:Label id=msg runat="server"></asp:Label><br>

```

The authenticated user's name can then be passed to the user control to provide additional customization. To pass the user name from the calling page, you need to make only a few modifications. First, in the declaration of the user control tag, add an *id* attribute so that the control can be referenced programmatically:

```
<PureASP:Simple id="menu" runat="server"></PureASP:Simple>
```

Then, it is possible to access the new public property on the calling page by using the following C# syntax:

```
menu.userName = "test";
```

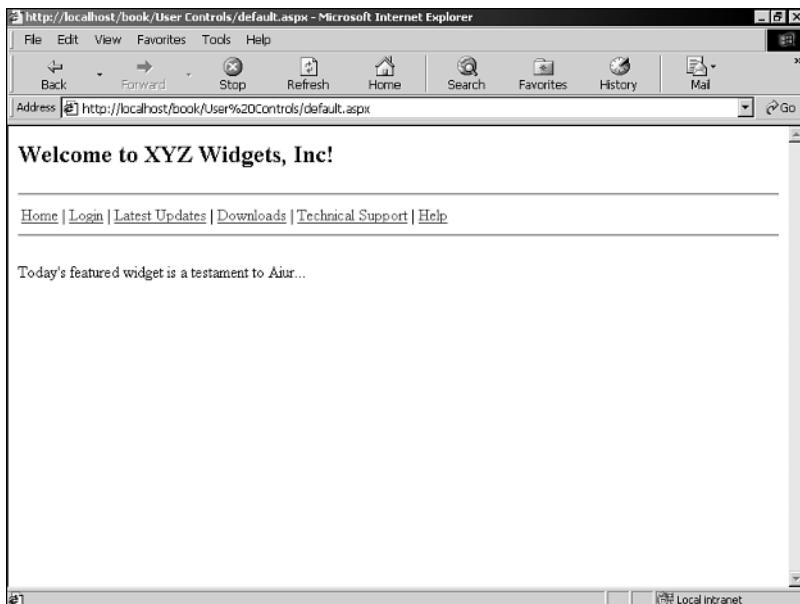
or by using the following Visual Basic.NET syntax:

```
menu.userName = "test"
```

Alternatively, you can set properties from within the user control tag itself:

```
<PureASP:Simple id="menu" userName="test" runat="server"></PureASP:Simple>
```

Figure 10.2 shows how the page appears when it is customized with the user name.

**Figure 10.2**

Customizing content with User controls.

If you need better control over how the public property is set, you can use a more formal property definition. The code in Listing 10.5 provides the same functionality for adding a property as in Listings 10.3 and 10.4. However, if you needed to perform validation on a property, such as verifying that the UserName is less than 20 characters, you could either throw an error instead of setting the value, or even only store the first 20 characters of the name.

Listing 10.5 Using Properties in User Controls

```
private string userName;

public String UserName {
    get {
        return UserName;
    }
    set {
        userName = value;
    }
}
```

Adding Methods to a User Control

You add public methods to user controls the same way you add public properties. You simply create a public method in the user control and then reference it from the calling

page. For example, to provide a function inside the Simple.ascx user control that adds two integers, place the function in a script block with the public attribute, as shown in Listing 10.6.

Listing 10.6 Adding a Public Method to a User Control

```
public int add( int i, int j )
{
    return i + j;
}
```

Then, on the calling page, the public method can be accessed by using the following line of code:

```
int sum = header.add(1234, 4321);
```

Fragment Caching

The Microsoft .NET framework provides a way to cache the output of a user control separately from the rest of the page. This is called *fragment caching*.

Enabling fragment caching for a user control is simple: All you need to do is add the `OutputCache` directive to the top of the user control. You can enable fragment caching simply by using the `OutputCache` page directive as seen in the first line of code in Listing 10.7. The `Duration` attribute of the `OutputCache` page directive specifies the amount of time the control is cached and is specified in seconds.

Listing 10.7 Enabling Fragment Caching

```
<%@ OutputCache Duration="120" %>
<!--
This user control is designed to simply output
a page header to be used on every page.
-->

<h2>Welcome to XYZ Widgets, Inc!</h2>
<hr>
<table>
<tr>
    <td><a href="">Home</a> |</td>
    <td><a href="">Login</a> |</td>
    <td><a href="">Latest Updates</a> |</td>
    <td><a href="">Downloads</a> |</td>
    <td><a href="">Technical Support</a> |</td>
    <td><a href="">Help</a></td>
</tr>
</table>
<hr>
```


CHAPTER 11

Working with ASP.NET List Controls

This chapter covers the ASP.NET list controls: the Repeater, DataList, and DataGrid controls. List Controls provide a way to easily render cross-browser Hypertext Markup Language (HTML) that contains the contents of a data source. In the past, in order to display the contents of a multi-row data source, you had to loop through the data manually and render the data in HTML within the loop. This cross-browser functionality is built into the list controls to provide a way for developers to create pages more efficiently and at the same time generate code that is clean and easy to read. This chapter discusses the list controls in detail, and it provides practical real-world examples for each.

Binding a data source to the list controls is a major part of the controls, and it is also discussed in this chapter.

Binding Data to List Controls

An ASP.NET list control can be bound to any data source that implements the `IEnumerable`, `ICollection`, or `IListSource` interfaces (for example, `DataView`, `OleDbDataReader`, `SqlDataReader`, `Hashtable`, and `ArrayList`). eXtensible Markup Language (XML) data can also be bound to the list controls, to fill a `DataView` object with its data and then bind the List Control to the `DataView` object. In addition, you can write custom classes to implement `IEnumerable` and allow those classes to be bound to the ASP.NET List Controls as well (see Table 11.1).

 **NOTE**

Note that the names of the `System.Data` objects have changed dramatically since Beta 1. Some of the changes are as follows:

Beta 1 Name	Current Name
<code>System.Data.SqlClient</code>	<code>System.Data.SqlClient</code>
<code>System.Data.OleDb</code>	<code>System.Data.OleDb</code>
<code>ADOConnection</code>	<code>OleDbConnection</code>
<code>ADODataReader</code>	<code>OleDbDataReader</code>
<code>ADOCmd</code>	<code>OleDbCommand</code>
<code>ADODataset</code>	<code>OleDbDataAdapter</code>

Table 11.1 Common Classes That Implement `IEnumerable`

Class	Description
<code>DataView</code>	Provides a customized view of a data table that supports sorting, filtering, searching, editing, and navigation
<code>OleDbDataReader</code>	Provides a forward-only reader for a stream of data records from an object linking and embedding database (OLE DB) data source
<code>SqlDataReader</code>	Provides a forward-only reader for a stream of data records from a Structured Query Language (SQL) data source
<code>Hashtable</code>	Contains a collection of key-and-value pairs
<code>SortedList</code>	Contains a collection of key-and-value pairs that are sorted based on the key
<code>ArrayList</code>	Contains an array that can dynamically change in size

Because the list controls support all classes that implement the `IEnumerable` interface, the process of binding the controls to the different classes is pretty much the same for all the classes. The following sections describes this process.

The `DataSource` Property

To bind a data source to a list control, you must first tell the list control what data source you want to bind it to. You do this with the `DataSource` property, whose syntax is as follows:

```
[ListControl].DataSource = [data source]
```

The `.DataBind()` Method

After the data source has been specified, the actual binding of the data needs to take place. You do this by calling the `DataBind()` method, which performs the following steps:

1. It enumerates the data list's bound data source.
2. It creates the list control's items.
3. It populates each item with data from the data source.
4. If state management is enabled, the `DataBind()` method stores the information needed to re-create the list control.

NOTE

Because the data source is not enumerated until the `DataBind()` method is invoked, the developer has the power to decide when the enumeration occurs by choosing where to call the `DataBind()` method. This helps to eliminate the need to make frequent trips to the data source when you need the information throughout the pages life.

Binding Expressions

All data binding expressions are contained within the `<%# %>` characters. These expressions are evaluated not when the page is processed (as is the case with `Response.Write`), but rather when the `DataBind()` method is invoked. The following is an example of a data-binding expression that inserts a date-formatted value from the data source:

```
<%# String.Format("{0:d}", ((DataRowView)Container.DataItem)["DateValue"]) %>
```

One of the problems with using the data binding expressions is that for the framework to know what object is being represented by `Container.DataItem`, it must cast the object. *Casting* is the process of converting an object of one type to another (as in the preceding example). The value of the `DateValue` field in the bound data source is retrieved by using the following code snippet:

```
Container.DataItem["DateValue"]
```

However, for the container to know what type of data source the `DateValue` object belongs to, you must cast the container like this:

```
((DataRowView)Container.DataItem)["DateValue"]
```

To make matters worse, in order to use the return value from a statement such as this one, you must often cast it as well. This means that the data binding expression would require two type casts.

To help ease the development and eliminate the need to perform much of the casting operations, ASP.NET provides the `DataBinder.Eval()` method. The following statement performs the same operation as the statement in the previous section:

```
<%# DataBinder.Eval(Container.DataItem, "DateValue", "{0:d}") %>
```

 **NOTE**

The `DataBinder.Eval()` method uses a Microsoft .NET feature known as *reflection*, which allows you to discover type information at runtime, to evaluate the bound data. This causes a slight performance degradation compared to using the direct data binding expression described earlier in this section. If performance is a major issue for your application, you might want to stay away from using `DataBinder.Eval()`.

Binding Examples

As discussed earlier in this chapter, the list controls can be bound to any data source that implements the `IEnumerable` interface. This section looks at some examples of binding different types of data sources to the list controls.

Binding a List Control to an ArrayList

One of the simplest types that can be bound to the list controls is an `ArrayList` object. `ArrayLists` are ideal for binding to the `Repeater`, `CheckBoxList`, and `RadioButtonList` controls because they typically hold data that contains a small number of fields. However, because array lists are so flexible, they can contain a wide assortment of object types, and therefore they are also practical for use with the `DownList` and `GridView` controls.

The example in Listings 11.1 and 11.2 shows how to bind an `ArrayList` with a `RadioButtonList` control (see Figure 11.1). You bind the control to the `ArrayList` by simply setting the `DataSource` property of the control to the `ArrayList` itself. When the `DataBind()` method is invoked, each item in the `ArrayList` is enumerated, and a new radio button will be added to the list control.

Listing 11.1 Binding a List Control to an ArrayList (C#)

```
<% @Page Language="C#" %>

<script language="C#" runat="server">

void Page_Load(Object Sender, EventArgs E) {

    if (!IsPostBack) {

        ArrayList values = new ArrayList();

        values.Add ("Red");
        values.Add ("Blue");
        values.Add ("Orange");
        values.Add ("Yellow");
        values.Add ("Green");
        values.Add ("Purple");

    }
}
```

Listing 11.1 continued

```
MyRadioButtonList.DataSource = values;
MyRadioButtonList.DataBind();
}
}

void Submit_Click(Object src, EventArgs e) {
    if (MyRadioButtonList.SelectedIndex > -1) {
        lblMessage.Text = "<b>You selected:</b> " +
            MyRadioButtonList.SelectedItem.Text;
    }
}

</script>

<HTML>
<HEAD>
    <TITLE>Simple RadioButtonList Example</TITLE>
</HEAD>
<BODY>

<h1>Simple RadioButtonList Example</h1>
<hr>

<form runat="server" id=form1 name=form1>

<asp:RadioButtonList id=MyRadioButtonList runat="server"/>

<br>
<br>

<asp:button id="btnSubmit" runat="server"
    Text="Submit" onclick="Submit_Click"/>

<br>
<br>

<asp:label id="lblMessage" runat="server"/>

</form>

</BODY>
</HTML>
```

Listing 11.2 Binding a List Control to an ArrayList (Visual Basic.NET)

```
<% @Page Language="VB" %>

<script language="VB" runat="server">

Sub Page_Load(Sender As Object, E As EventArgs)
    If Not IsPostBack Then
        Dim values As ArrayList = new ArrayList
        values.Add ("Red")
        values.Add ("Blue")
        values.Add ("Orange")
        values.Add ("Yellow")
        values.Add ("Green")
        values.Add ("Purple")

        MyRadioButtonList.DataSource = values
        MyRadioButtonList.DataBind()
    End If
End Sub

Sub Submit_Click(src As Object, e As EventArgs)
    If MyRadioButtonList.SelectedIndex > -1 Then
        lblMessage.Text = "<b>You selected:</b> " + _
            MyRadioButtonList.SelectedItem.Text
    End If
End Sub

</script>

<HTML>
<HEAD>
    <TITLE>Simple RadioButtonList Example</TITLE>
</HEAD>
<BODY>

<h1>Simple RadioButtonList Example</h1>
<hr>

<form runat="server" id=form1 name=form1>
```

Listing 11.1 continued

```
<asp:RadioButtonList id=MyRadioButtonList runat="server" />

<br>
<br>

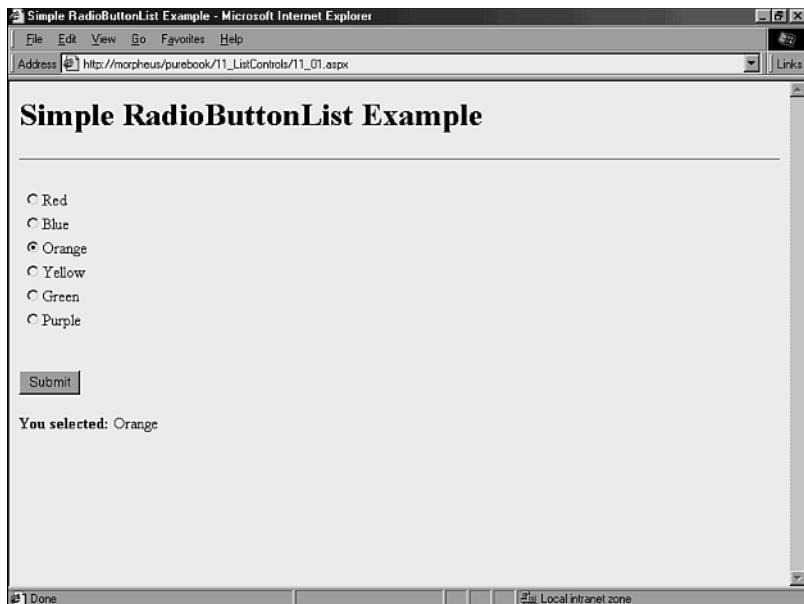
<asp:button id="btnSubmit" runat="server"
    Text="Submit" onclick="Submit_Click" />

<br>
<br>

<asp:label id="lblMessage" runat="server" />

</form>

</BODY>
</HTML>
```

**Figure 11.1**

An example of binding a list control to an arraylist.

Binding a List Control to a Hashtable

The example in Listing 11.3 and 11.4 show how to bind a hashtable to a list control (see Figure 11.2). You can use a hashtable to store name/value pairs and you can bind a hashtable to a list control by simply setting a list control's `DataSource` property to

the instance of the Hashtable object and invoking the DataBind() method, as with binding to an arraylist.

Listing 11.3 Binding a List Control to a Hashtable (C#)

```
<% @Page Language="C#" %>

<script language="C#" runat="server">

    void Page_Load(Object sender, EventArgs e) {

        if (!Page.IsPostBack) {

            Hashtable h = new Hashtable();
            h.Add ("Product One", "13");
            h.Add ("Product Two", "11");
            h.Add ("Product Three", "2");
            h.Add ("Product Four", "23");

            MyDataGrid.DataSource = h;
            MyDataGrid.DataBind();
        }
    }

</script>

<HTML>
<HEAD>
    <TITLE>Data Binding to a Hashtable</TITLE>
</HEAD>
<BODY>

<h1>Data Binding to a Hashtable</h1>
<hr>

<asp:DataGrid id="MyDataGrid" runat="server" AutoGenerateColumns="false">

    <Columns>

        <asp:TemplateColumn HeaderText="Product">

            <ItemTemplate>
                <%# ((DictionaryEntry)Container.DataItem).Key %>
            </ItemTemplate>

        </asp:TemplateColumn>

        <asp:TemplateColumn HeaderText="Quantity">
```

Listing 11.3 continued

```
<ItemTemplate>
    <%# ((DictionaryEntry)Container.DataItem).Value %>
</ItemTemplate>

</asp:TemplateColumn>

</Columns>

</asp:DataGrid>

</BODY>
</HTML>
```

Listing 11.4 Binding a List Control to a Hashtable (Visual Basic.NET)

```
<% @Page Language="VB" %>

<script language="VB" runat="server">

Sub Page_Load(sender As Object, e As EventArgs)

    If Not Page.IsPostBack Then

        Dim h As Hashtable = new Hashtable
        h.Add ("Product One", "13")
        h.Add ("Product Two", "11")
        h.Add ("Product Three", "2")
        h.Add ("Product Four", "23")

        MyDataGrid.DataSource = h
        MyDataGrid.DataBind

    End If

End Sub

</script>

<HTML>
<HEAD>
    <TITLE>Data Binding to a Hashtable</TITLE>
</HEAD>
<BODY>

<h1>Data Binding to a Hashtable</h1>
<hr>

<asp:DataGrid id="MyDataGrid" runat="server" AutoGenerateColumns="false">
```

Listing 11.4 continued

```
<Columns>

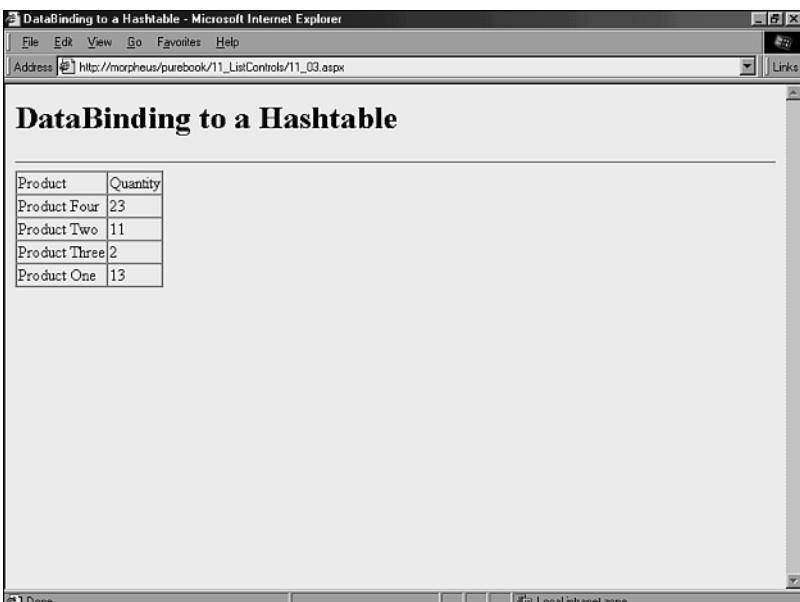
    <asp:TemplateColumn HeaderText="Product">
        <ItemTemplate>
            <%# (Container.DataItem).Key %>
        </ItemTemplate>
    </asp:TemplateColumn>

    <asp:TemplateColumn HeaderText="Quantity">
        <ItemTemplate>
            <%# (Container.DataItem).Value %>
        </ItemTemplate>
    </asp:TemplateColumn>

</Columns>

</asp:DataGrid>

</BODY>
</HTML>
```

**Figure 11.2**

An example of binding a list control to a hashtable.

Binding a List Control to a Dataview

DataView objects are commonly bound to list controls. A dataview is very close to an ADO recordset and is excellent at storing large amounts of data with multiple fields. Dataviews also implement the `ICollection` interface and therefore can be bound to list controls in the same fashion as arraylists and hashtables. (ADO.NET is discussed in detail in Chapter 13, “Data Access with ADO.NET.”)

Listings 11.5 and 11.6 show how to bind a `DataGrid` control to a dataview (see Figure 11.3). (The `DataGrid` control is discussed later in this chapter.)

Listing 11.5 Binding a List Control to a DataView (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="System.Data.SqlClient" %>
<% @Import Namespace="System.Data" %>

<script language="C#" runat="server">

void Page_Load(Object Src, EventArgs E) {

    // Create Instance of Connection and Command Object
    SqlConnection cn = new
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind");
    SqlDataAdapter adapter = new SqlDataAdapter("select * from employees", cn);

    DataSet myDataSet = new DataSet();
    adapter.Fill(myDataSet, "employees");

    mygrid.DataSource = myDataSet.Tables["employees"];
    mygrid.DataBind();

}

</script>

<HTML>
<HEAD>
    <TITLE>Binding to a DataView Example</TITLE>
</HEAD>
<BODY>

<h1>Binding to a DataView Example</h1>
<HR>

<asp:datagrid id="mygrid" runat="server" />

</BODY>
</HTML>
```

Listing 11.6 Binding a List Control to a DataView (Visual Basic.NET)

```
<% @Page Language="VB" %>
<% @Import Namespace="System.Data.SqlClient" %>
<% @Import Namespace="System.Data" %>

<script language="VB" runat="server">

Sub Page_Load(Sender As Object, E As EventArgs)

    'Create Instance of Connection and Command Object
    Dim cn As SqlConnection = New_
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind")
    Dim adapter As SqlDataAdapter = New_
        SqlDataAdapter("select * from employees", cn)

    Dim myDataSet As DataSet = new DataSet
    adapter.Fill(myDataSet, "employees")

    mygrid.DataSource = myDataSet.Tables("employees")
    mygrid.DataBind

End Sub

</script>

<HTML>
<HEAD>
    <TITLE>Binding to a DataView Example</TITLE>
</HEAD>
<BODY>

<h1>Binding to a DataView Example</h1>
<HR>

<asp:datagrid id="mygrid" runat="server" />

</BODY>
</HTML>
```

Binding a List Control to an XML Data Source

At times, you might want to bind a list control to XML data. You can do this by filling a dataview with data from an XML file and binding that dataview to the list control. The dataview is populated with the XML data by using the `ReadXml` method of the dataview (which is discussed more in Chapter 12, “Working with ASP.NET Validation Controls”), which accepts a `StreamReader` object. The `StreamReader` object is populated with a `FileStream` object that reads the XML data.

The screenshot shows a Microsoft Internet Explorer window with the title "Binding to a DataView Example - Microsoft Internet Explorer". The address bar shows the URL "http://morpheus/purebook/11_ListControls/11_05.aspx". The main content area displays a DataGrid with the following data:

EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate	HireDate	Address	City	Region	Phone
1	Davolio	Nancy	Sales Representative	Ms.	12/8/1948 12:00:00 AM	5/1/1992 12:00:00 AM	507 - 20th Ave. E Apt 2A	Seattle	WA	98

Figure 11.3

An example of binding a list control to a Dataview.

For example, the XML data in Listing 11.7 contains a list of products, and for each product, it contains ProductID, CategoryID, ProductName, ProductDescription, UnitPrice, and Manufacturer properties. The code that binds the data from Listing 11.7 to a DataGrid object is shown in Listings 11.8 and 11.9. (see Figure 11.4). Note that in order for the examples to work, you must make sure that the XML file exists.

Listing 11.7 XML Data that is bound to the List Control

```
<DocumentElement>
  <Products>
    <ProductID>1001</ProductID>
    <CategoryID>1</CategoryID>
    <ProductName>Chocolate City Milk</ProductName>
    <ProductDescription>
      Chocolate City Milk Description
    </ProductDescription>
    <UnitPrice>2</UnitPrice>
    <Manufacturer>Chocolate City</Manufacturer>
  </Products>
  <Products>
    <ProductID>1002</ProductID>
    <CategoryID>1</CategoryID>
    <ProductName>Bessie Brand 2% Milk</ProductName>
    <ProductDescription>
```

Listing 11.7 continued

```
Bessie Brand 2% Milk Description
</ProductDescription>
<UnitPrice>1.19</UnitPrice>
<Manufacturer>Milk Factory</Manufacturer>
</Products>
<Products>
<ProductID>1003</ProductID>
<CategoryID>1</CategoryID>
<ProductName>Funny Farms Milk</ProductName>
<ProductDescription>
    Funny Farms Whole Milk Description
</ProductDescription>
<UnitPrice>1.29</UnitPrice>
<Manufacturer>Funny Farms</Manufacturer>
</Products>
<Products>
<ProductID>1005</ProductID>
<CategoryID>1</CategoryID>
<ProductName>Marigold Whole Milk</ProductName>
<ProductDescription>
    Marigold Whole Milk Description
</ProductDescription>
<UnitPrice>1.39</UnitPrice>
<Manufacturer>Marigold Meadows</Manufacturer>
</Products>
<Products>
<ProductID>2001</ProductID>
<CategoryID>2</CategoryID>
<ProductName>Fruity Pops</ProductName>
<ProductDescription>
    Fruity Pops Description
</ProductDescription>
<UnitPrice>4.07</UnitPrice>
<Manufacturer>River Mills</Manufacturer>
</Products>
</DocumentElement>
```

Listing 11.8 Binding a List Control to an XML Data Source (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.IO" %>

<script language="C#" runat="server">

    void Page_Load(Object src, EventArgs e) {

        DataSet ds = new DataSet();
```

Listing 11.8 continued

```
FileStream fs = new FileStream(
    Server.MapPath("mydata.xml"), FileMode.Open, FileAccess.Read );
StreamReader reader = new StreamReader( fs );
ds.ReadXml( reader );

fs.Close();
DataView Source = new DataView( ds.Tables[0] );
MyDataGrid.DataSource = Source;
MyDataGrid.DataBind();

}

</script>

<HTML>
<HEAD>
    <TITLE>Data Binding to XML</TITLE>
</HEAD>
<BODY>

<h1>Data Binding to XML</h1>
<hr>

<asp:DataGrid id="MyDataGrid" runat="server" />

</BODY>
</HTML>
```

Listing 11.9 Binding a List Control to an XML Data Source (Visual Basic.NET)

```
<% @Page Language="VB" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.IO" %>

<script language="VB" runat="server">

Sub Page_Load(Sender As Object, E As EventArgs)

    Dim ds As DataSet = new DataSet

    Dim fs As FileStream = new FileStream( _
        Server.MapPath("mydata.xml"), FileMode.Open, FileAccess.Read )
    Dim reader As StreamReader = new StreamReader( fs )
    ds.ReadXml( reader )
```

Listing 11.9 continued

```
fs.Close()
Dim Source As DataView = new DataView( ds.Tables(0) )
MyDataGrid.DataSource = Source
MyDataGrid.DataBind

End Sub

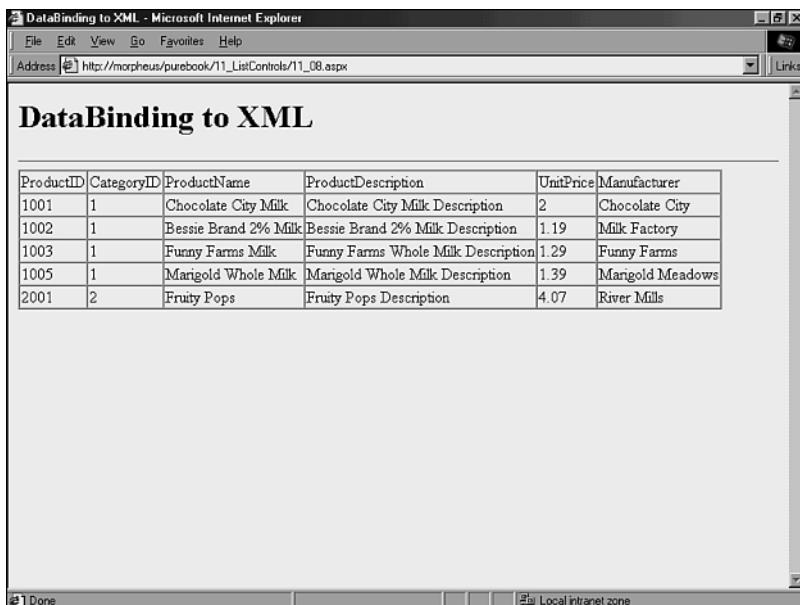
</script>

<HTML>
<HEAD>
    <TITLE>Data Binding to XML</TITLE>
</HEAD>
<BODY>

<h1>Data Binding to XML</h1>
<hr>

<asp:DataGrid id="MyDataGrid" runat="server" />

</BODY>
</HTML>
```



DataBinding to XML

ProductID	CategoryID	ProductName	ProductDescription	UnitPrice	Manufacturer
1001	1	Chocolate City Milk	Chocolate City Milk Description	2	Chocolate City
1002	1	Bessie Brand 2% Milk	Bessie Brand 2% Milk Description	1.19	Milk Factory
1003	1	Funny Farms Milk	Funny Farms Whole Milk Description	1.29	Funny Farms
1005	1	Marigold Whole Milk	Marigold Whole Milk Description	1.39	Marigold Meadows
2001	2	Fruity Pops	Fruity Pops Description	4.07	River Mills

Figure 11.4

An example of binding a list control to an XML data source.

The Repeater List Control

The most basic list control is the Repeater control. As its name suggests, the Repeater control is used to display a collection of items in a repeating list. The items are rendered in the list based on templates.

Repeater Control Templates

As shown in Table 11.2, a number of templates are available to the Repeater control. You can use these templates to customize how a list appears.

Table 11.2 Templates Available to the Repeater Control

Template Name	Description
HeaderTemplate	Defines the content of the list's header
ItemTemplate	Defines the layout of items in the list
AlternatingItemTemplate	Defines the layout of every other item in the list
SeparatorTemplate	Defines the content in between items in the list
FooterTemplate	Defines the content of the list's footer

Of the Repeater control's templates, at the minimum developers must define ItemTemplate. Without a definition for ItemTemplate, there is no way for the Repeater control to know how to display the bound data.

For example, say that a Repeater control contains the definitions for all five of the templates and it is bound to a data source that has five records. When the DataBind method is invoked, first the content defined in HeaderTemplate is rendered. Then, every record in the data source with an odd index renders the content defined in AlternatingItemTemplate, and every record with an even index renders the content defined in ItemTemplate. After all the records in the data source are enumerated, the content defined in FooterTemplate is rendered at the end of the list. Figure 11.5 shows the content that would be rendered in the Repeater control in this example.

Using Repeater Controls

As you know, Repeater controls are used to display collections of data in a repeating list. For example, say you need to display a list of links. The Repeater control is a good control to choose for this because it does the job without providing additional functionality that you do not need (as would be the case with the `DownList` and `DataGrid` controls, which are discussed later in this chapter). The example shown in Figure 11.6 demonstrates using the Repeater control to create a list of links.

Listing 11.10 shows the XML data that is used to create the example shown in Figure 11.6. The data contains two elements, the `Text` element, which is the displayed text, and the `Link` element, which is the URL that will be linked to the text.

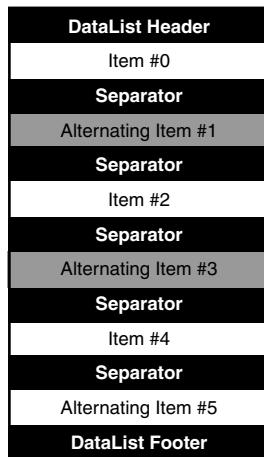


Figure 11.5

Content rendered by the Repeater control.

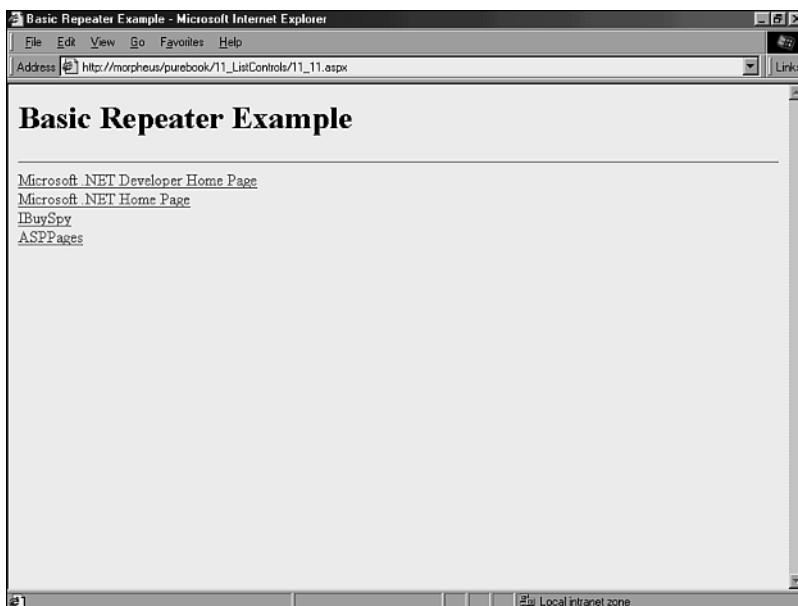


Figure 11.6

A simple Repeater control example.

Listing 11.10 XML Data Containing a Collection of Links

```
<DocumentElement>
  <Links>
    <Text>Microsoft .NET Developer Home Page</Text>
```

Listing 11.10 continued

```

<Link>http://msdn.microsoft.com/net</Link>
</Links>
<Links>
    <Text>Microsoft .NET Home Page</Text>
    <Link>http://www.microsoft.com/net</Link>
</Links>
<Links>
    <Text>IBuySpy</Text>
    <Link>http://www.ibuyspy.com</Link>
</Links>
<Links>
    <Text>ASPPages</Text>
    <Link>http://www.asppages.com</Link>
</Links>
</DocumentElement>

```

The Repeater control in this example defines only the `ItemTemplate` template. If you wanted to add a header item to the list of links, you could define a `HeaderTemplate` template. Similarly, if you wanted to add a separator between the links or if you wanted to add a footer to the bottom of the list, you could add definitions for the `SeparatorTemplate` and `FooterTemplate` templates.

To create the links, you add the `HyperLink` web control to the `ItemTemplate` definition. For every record in the data source, an instance of the `HyperLink` control will be generated and will be bound to that record's data. The code for this example is shown in Listings 11.11 and 11.12.

Listing 11.11 A Basic Repeater Control Example (C#)

```

<% @Page Language="C#" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.IO" %>

<script language="C#" runat="server">

void Page_Load(Object src, EventArgs e) {

    DataSet ds = new DataSet();

    FileStream fs = new FileStream( Server.MapPath("links.xml"),
        FileMode.Open, FileAccess.Read );
    StreamReader reader = new StreamReader( fs );
    ds.ReadXml( reader );

    fs.Close();
    DataView Source = new DataView( ds.Tables[0] );
    MyRepeater.DataSource = Source;
    MyRepeater.DataBind();
}

```

Listing 11.11 continued

```
}

</script>

<HTML>
<HEAD>
    <TITLE>Basic Repeater Example</TITLE>
</HEAD>
<BODY>

<h1>Basic Repeater Example</h1>
<hr>

<asp:Repeater id="MyRepeater" runat="server">

    <ItemTemplate>
        <asp:Hyperlink runat="server"
            NavigateUrl='<%# DataBinder.Eval(Container.DataItem, "Link") %>'
            Text='<%# DataBinder.Eval(Container.DataItem, "Text") %>'/>
        <br>
    </ItemTemplate>

</asp:Repeater>

</BODY>
</HTML>
```

Listing 11.12 A Basic Repeater Control Example (Visual Basic.NET)

```
<% @Page Language="VB" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.IO" %>

<script language="VB" runat="server">

    Sub Page_Load(Sender As Object, E As EventArgs)

        Dim ds As DataSet = new DataSet

        Dim fs As FileStream = new FileStream( Server.MapPath("links.xml"), _
            FileMode.Open, FileAccess.Read )
        Dim reader As StreamReader = new StreamReader( fs )
        ds.ReadXml( reader )

        fs.Close()
        Dim Source As DataView = new DataView( ds.Tables(0) )
        MyRepeater.DataSource = Source
        MyRepeater.DataBind

    End Sub
</script>
```

Listing 11.12 continued

```

End Sub

</script>

<HTML>
<HEAD>
    <TITLE>Basic Repeater Example</TITLE>
</HEAD>
<BODY>

<h1>Basic Repeater Example</h1>
<hr>

<asp:Repeater id="MyRepeater" runat="server">

    <ItemTemplate>
        <asp:Hyperlink runat="server"
            NavigateUrl='<%# DataBinder.Eval(Container.DataItem, "Link") %>'
            Text='<%# DataBinder.Eval(Container.DataItem, "Text") %>' />
        <br>
    </ItemTemplate>

</asp:Repeater>

</BODY>
</HTML>

```

The DataList List Control

Like the Repeater control, the DataList control is a template-driven list control. However, the DataList control provides additional functionality such as the ability to support styles, display items in columns, select items, and edit in place.

DataList Control Templates

Table 11.3 lists the templates that are available to the DataList control.

Table 11.3 Templates Available to the DataList Control

Template Name	Description
HeaderTemplate	Defines the content of the list's header
ItemTemplate	Defines the layout of items in the list
AlternatingItemTemplate	Defines the layout of every other item in the list
SeparatorTemplate	Defines the content in between items in the list
FooterTemplate	Defines the content of the list's footer
SelectedItemTemplate	Defines the content of the selected item in the list

As with the Repeater control, the ItemTemplate template is required and all the other templates are optional.

Using DataList Controls

Creating a page that lists products is an ideal task for the DataList control because of its ability to list the items in columns. In the example shown in Listings 11.13 and 11.14, the products from the XML data shown in Listing 11.7 are displayed (see Figure 11.7).

Listing 11.13 A Basic DataList Control Example (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.IO" %>

<script language="C#" runat="server">

    void Page_Load(Object src, EventArgs e) {

        DataSet ds = new DataSet();

        FileStream fs = new FileStream( Server.MapPath("mydata.xml"),
            FileMode.Open, FileAccess.Read );
        StreamReader reader = new StreamReader( fs );
        ds.ReadXml( reader );

        fs.Close();
        DataView Source = new DataView( ds.Tables[0] );
        MyDataList.DataSource = Source;
        MyDataList.DataBind();

    }

</script>

<HTML>
<HEAD>
    <TITLE>Basic DataList Example</TITLE>
</HEAD>
<BODY>

<asp:DataList id="MyDataList" runat="server">

    <ItemTemplate>

        <asp:Label runat="server"
            Font-Bold="true"
```

Listing 11.13 continued

```

    text='<%# DataBinder.Eval(Container.DataItem, "ProductName") %>' />
<br>
ProductID:
<asp:Label runat="server"
    text='<%# DataBinder.Eval(Container.DataItem, "ProductID") %>' />
<br>
Manufacturer:
<asp:Label runat="server"
    text='<%# DataBinder.Eval(Container.DataItem, "Manufacturer") %>' />
<br>
<asp:Label runat="server" font-italic="true"
    text=
        '<%# DataBinder.Eval(Container.DataItem, "ProductDescription") %>'
    />
<br><br>

</ItemTemplate>

</asp:DataList>

</BODY>
</HTML>

```

Listing 11.14 A Basic DataList Control Example (Visual Basic.NET)

```

<% @Page Language="VB" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.IO" %>

<script language="VB" runat="server">

Sub Page_Load(Sender As Object, E As EventArgs)

    Dim ds As DataSet = new DataSet

    Dim fs As FileStream = new FileStream( Server.MapPath("mydata.xml"), _
 FileMode.Open, FileAccess.Read )
    Dim reader As StreamReader = new StreamReader( fs )
    ds.ReadXml( reader )

    fs.Close()
    Dim Source As DataView = new DataView( ds.Tables(0) )
    MyDataList.DataSource = Source
    MyDataList.DataBind

End Sub

```

Listing 11.14 continued

```
</script>

<HTML>
<HEAD>
    <TITLE>Basic DataList Example</TITLE>
</HEAD>
<BODY>

<asp:DataList id="MyDataList" runat="server">

    <ItemTemplate>

        <asp:Label runat="server"
            Font-Bold="true"
            text='<%# DataBinder.Eval(Container.DataItem, "ProductName") %>' />
        <br>
        ProductID:
        <asp:Label runat="server"
            text='<%# DataBinder.Eval(Container.DataItem, "ProductID") %>' />
        <br>
        Manufacturer:
        <asp:Label runat="server"
            text='<%# DataBinder.Eval(Container.DataItem, "Manufacturer") %>' />
        <br>
        <asp:Label runat="server" font-italic="true"
            text=
                '<%# DataBinder.Eval(Container.DataItem, "ProductDescription") %>'
            />
        <br><br>

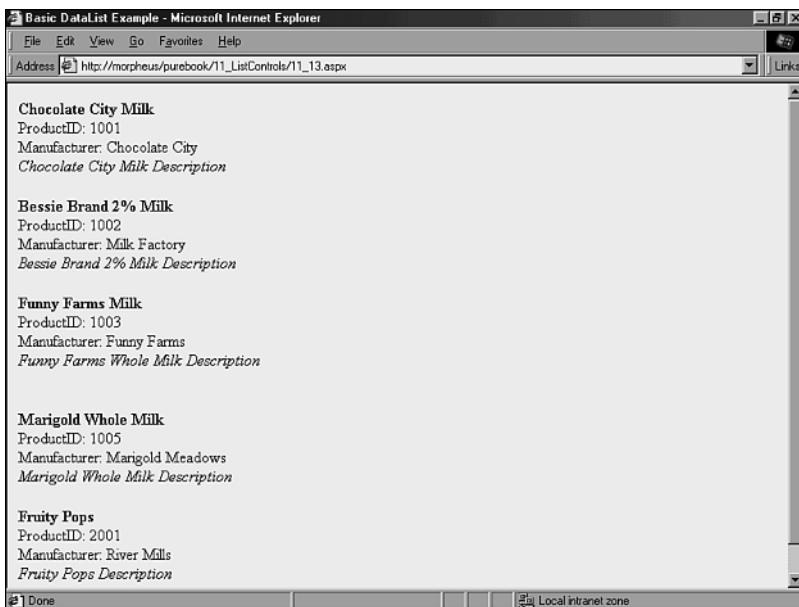
    </ItemTemplate>

</asp:DataList>

</BODY>
</HTML>
```

Formatting a DataList Control with Styles

One of the features that the `DataList` control supports is the ability to format a list with style properties. This makes it possible for you to set style attributes for the list by simply setting the built-in properties. As shown in Table 11.4, a number of properties are available for setting the styles of different items in a `DataList` control.

**Figure 11.7**

A basic DataList control example.

Table 11.4 Style Properties Available to the DataList Control

Property	Description
AlternatingItemStyle	Defines the style of all alternating items in the list
EditItemStyle	Defines the style of the item in the list that is currently being edited
FooterStyle	Defines the style of the footer in the list
HeaderStyle	Defines the style of the header in the list
ItemStyle	Defines the style of all items in the list
SelectedItemStyle	Defines the style of the item in the list that is currently selected
SeparatorStyle	Defines the style of all separator items in the list

Each of the properties listed in Table 11.4 is of the `TableItemStyle` type. You can set individual style attributes by using the properties of the `TableItemStyle` class, which are listed in Table 11.5.

Table 11.5 Properties of the TableItemStyle Class

Property	Description
BackColor	Sets the background color of the current item.
BorderColor	Sets the border color of the current item.
BorderStyle	Sets the border style of the current item. The following options are available: <ul style="list-style-type: none">• Dashed• Dotted• Double• Groove• Inset• None• OutSet• Ridge• Solid
BorderWidth	Sets the border width of the current item.
CssClass	Sets the <code>css</code> class of the current item.
Font-Bold	Sets the <code>Bold</code> attribute of the current item. The following options are available: <ul style="list-style-type: none">• True• False
Font-Italic	Sets the <code>Italic</code> attribute of the current item. The following options are available: <ul style="list-style-type: none">• True• False
Font-Underline	Sets the <code>Underline</code> attribute of the current item. The following options are available: <ul style="list-style-type: none">• True• False
FontSize	Sets the font size of the current item (for example, 10pt).
FontName	Sets the font of the current item (for example, Arial).
ForeColor	Sets the font color of the current item (for example, Red).
Height	Sets the height of the current item (for example, 100).
HorizontalAlign	Sets the horizontal alignment of the current item. The following options are available: <ul style="list-style-type: none">• Center• Justify• Left• Right
VerticalAlign	Sets the vertical alignment of the current item. The following options are available: <ul style="list-style-type: none">• Center• Justify• Left• Right

Table 11.5 continued

Property	Description
Width	Sets the width of the current item (for example, 100).
Wrap	Indicates whether text can wrap within the current item's cell. The following options are available: <ul style="list-style-type: none"> • True • False

By using the style attributes listed in Table 11.5, you can define the styles of a **DataList** control. The example shown in Listings 11.15 and 11.16 demonstrates how to use styles to customize the look of the **DataList** control introduced in Listings 11.13 and 11.14.

Listing 11.15 Applying Styles to a DataList Control (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.IO" %>

<script language="C#" runat="server">

void Page_Load(Object src, EventArgs e) {

    DataSet ds = new DataSet();

    FileStream fs = new FileStream( Server.MapPath("mydata.xml"),
        FileMode.Open, FileAccess.Read );
    StreamReader reader = new StreamReader( fs );
    ds.ReadXml( reader );

    fs.Close();
    DataView Source = new DataView( ds.Tables[0] );
    MyDataList.DataSource = Source;
    MyDataList.DataBind();

}

</script>

<HTML>
<HEAD>
    <TITLE>Formatting DataList with Styles Example</TITLE>
</HEAD>
<BODY>

<asp:DataList id="MyDataList" runat="server"
    ItemStyle-Font-Name="Arial"
    ItemStyle-Font-Size="10pt"
```

Listing 11.15 continued

```
ItemStyle-ForeColor="Red"
ItemStyle-HorizontalAlign="Left"
ItemStyle-BorderStyle="Double"
AlternatingItemStyle-BackColor="Khaki"
AlternatingItemStyle-ForeColor="blue"
AlternatingItemStyle-Underline="true"
AlternatingItemStyle-Height="100"
AlternatingItemStyle-HorizontalAlign="Right"
AlternatingItemStyle-BorderStyle="Dotted">

<ItemTemplate>

    <asp:Label runat="server"
        Font-Bold="true"
        text='<%# DataBinder.Eval(Container.DataItem, "ProductName") %>' />
    <br>
    ProductID:
    <asp:Label runat="server"
        text='<%# DataBinder.Eval(Container.DataItem, "ProductID") %>' />
    <br>
    Manufacturer:
    <asp:Label runat="server"
        text='<%# DataBinder.Eval(Container.DataItem, "Manufacturer") %>' />
    <br>
    <asp:Label runat="server" font-italic="true"
        text='<%#
            DataBinder.Eval(Container.DataItem, "ProductDescription") %>' />
    <br><br>

</ItemTemplate>

</asp:DataList>

</BODY>
</HTML>
```

Listing 11.16 Applying Styles to a DataList Control (Visual Basic.NET)

```
<% @Page Language="VB" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.IO" %>

<script language="VB" runat="server">

    Sub Page_Load(Sender As Object, E As EventArgs)
```

Listing 11.16 continued

```
Dim ds As DataSet = new DataSet

    Dim fs As FileStream = new FileStream( Server.MapPath("mydata.xml"), _
    FileMode.Open, FileAccess.Read )
    Dim reader As StreamReader = new StreamReader( fs )
    ds.ReadXml( reader )

    fs.Close()
    Dim Source As DataView = new DataView( ds.Tables(0) )
    MyDataList.DataSource = Source
    MyDataList.DataBind

End Sub

</script>

<HTML>
<HEAD>
    <TITLE>Formatting DataList with Styles Example</TITLE>
</HEAD>
<BODY>

<asp:DataList id="MyDataList" runat="server"
    ItemStyle-Font-Name="Arial"
    ItemStyle-Font-Size="10pt"
    ItemStyle-ForeColor="Red"
    ItemStyle-HorizontalAlign="Left"
    ItemStyle-BorderStyle="Double"
    AlternatingItemStyle-BackColor="Khaki"
    AlternatingItemStyle-ForeColor="blue"
    AlternatingItemStyle-Underline="true"
    AlternatingItemStyle-Height="100"
    AlternatingItemStyle-HorizontalAlign="Right"
    AlternatingItemStyle-BorderStyle="Dotted">

    <ItemTemplate>

        <asp:Label runat="server"
            Font-Bold="true"
            text='<%# DataBinder.Eval(Container.DataItem, "ProductName") %>' />
        <br>
        ProductID:
        <asp:Label runat="server"
            text='<%# DataBinder.Eval(Container.DataItem, "ProductID") %>' />
        <br>
        Manufacturer:
```

Listing 11.16 continued

```
<asp:Label runat="server"
    text='<%# DataBinder.Eval(Container.DataItem, "Manufacturer") %>' />
<br>
<asp:Label runat="server" font-italic="true"
    text=
        '<%# DataBinder.Eval(Container.DataItem, "ProductDescription") %>'
    />
<br><br>

</ItemTemplate>

</asp:DataList>

</BODY>
</HTML>
```

Creating a Columnar DataList Control

As mentioned previously, one of the powerful features of the `DataList` control is its ability to display items across a number of columns, in a columnar type of display. This feature is useful when you are creating pages for lists such as product listings and search results.

Two properties are primarily used to control how the `DataList` control displays the items across columns: `RepeatColumns` and `RepeatDirection`.

The RepeatColumns Property

The `RepeatColumns` property tells the `DataList` control across how many columns the items should span. If the `RepeatColumns` property is set to 2, the items in the `DataList` control will be rendered across two columns instead of one. Figures 11.8 and 11.9 show a `DataList` control displayed in one column and a `DataList` control displayed in two columns.

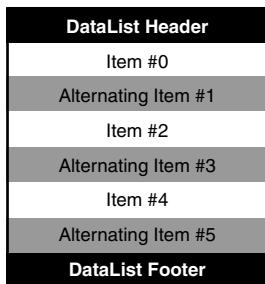


Figure 11.8

`DataList` control items rendered in one column.

DataList Header	
Item #0	Alternating Item #3
Alternating Item #1	Item #4
Item #2	Alternating Item #5
DataList Footer	

Figure 11.9

A `DataList` control with `RepeatColumns` set to 2.

The RepeatDirection Property

Whereas the `RepeatColumns` property indicates over how many columns to span the items, the `RepeatDirection` property tells the `DataList` control in which direction to populate the columns. The `RepeatDirection` property can have one of two values:

- **Vertical**—Columns are populated from top to bottom, and then from left to right.
- **Horizontal**—Columns are populated from left to right, and then from top to bottom.

When the `RepeatDirection` property is set to `Horizontal`, it fills the columns from left to right, and it adds rows when needed. If the `RepeatDirection` property is set to `Vertical`, it fills the columns from top to bottom, and then from left to right. This gives the developer complete control over how the items appear in the columns. Figures 11.10 and 11.11 show a `DataList` control with `RepeatDirection` set to `Vertical` and a `DataList` control with `RepeatDirection` set to `Horizontal`.

DataList Header	
Item #0	Alternating Item #3
Alternating Item #1	Item #4
↓ Item #2	↓ Alternating Item #5
DataList Footer	

Figure 11.10

A `DataList` control with `RepeatDirection` set to `Vertical`.

DataList Header	
Item #0	Alternating → Item #1
Item #2	Alternating → Item #3
Item #4	Alternating → Item #5
DataList Footer	

Figure 11.11

A `DataList` control with `RepeatDirection` set to `Horizontal`.

Selecting Items in a DataList Control

Another feature of the `DataList` control is the ability to select an individual item from a list. This makes the `DataList` control a good choice for creating navigational menus in which the currently selected item in the menu looks different from the other menu items.

The SelectedIndex Property

You select an item in the `DataList` control by setting the `SelectedIndex` property equal to the item's index, as in the following example:

```
MyDataList.SelectedIndex = 1;
```

The selected item's format is determined by the definition of the `SelectedItemTemplate` template. If `SelectedItemTemplate` is defined, the selected item will use this template instead of `ItemTemplate` or `AlternatingItemTemplate`. Styles are also available for the `SelectedItem` template; for instance, the following sets the background color for the selected item:

```
SelectedItem.BackColor="blue"
```

The example shown in Listings 11.17 and 11.18 is the same example as in Listings 11.13 and 11.14, but it takes the earlier example a step further by defining the `SelectedItemTemplate` template and selecting the second item in the `DataList` control.

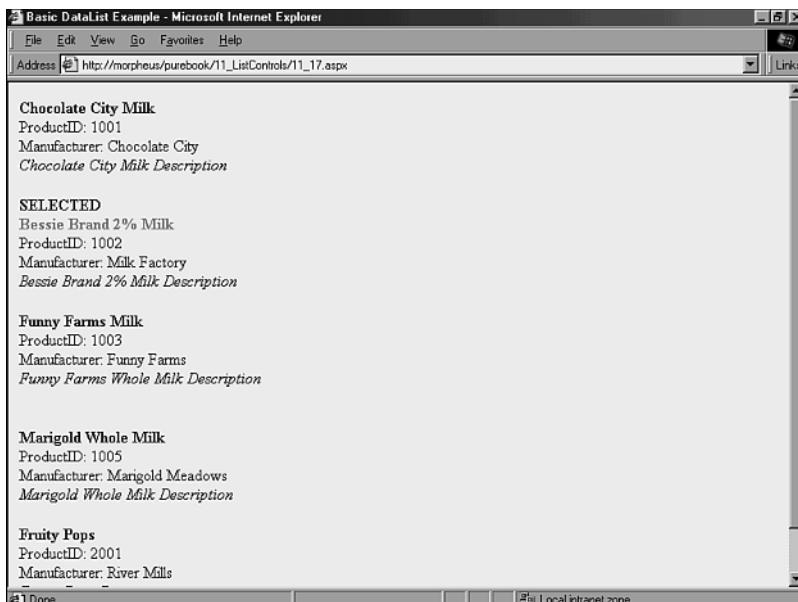


Figure 11.12

An example of selecting items in a `DataList` control.

Listing 11.17 Selecting Items in a DataList Control (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.IO" %>

<script language="C#" runat="server">

    void Page_Load(Object src, EventArgs e) {

        DataSet ds = new DataSet();

        FileStream fs = new FileStream( Server.MapPath("mydata.xml"),
            FileMode.Open, FileAccess.Read );
        StreamReader reader = new StreamReader( fs );
        ds.ReadXml( reader );

        fs.Close();
        DataView Source = new DataView( ds.Tables[0] );
        MyDataList.DataSource = Source;
        MyDataList.SelectedIndex = 1;
        MyDataList.DataBind();

    }

</script>

<HTML>
<HEAD>
    <TITLE>Selecting Items in a DataList</TITLE>
</HEAD>
<BODY>

<asp:DataList id="MyDataList" runat="server">

    <ItemTemplate>

        <asp:Label runat="server"
            Font-Bold="true"
            text='<%# DataBinder.Eval(Container.DataItem, "ProductName") %>' />
        <br>
        ProductID:
        <asp:Label runat="server"
            text='<%# DataBinder.Eval(Container.DataItem, "ProductID") %>' />
        <br>
        Manufacturer:
        <asp:Label runat="server"
            text='<%# DataBinder.Eval(Container.DataItem, "Manufacturer") %>' />
        <br>

    </ItemTemplate>

</asp:DataList>
```

Listing 11.17 continued

```
<asp:Label runat="server" font-italic="true"
    text='<%#
        DataBinder.Eval(Container.DataItem, "ProductDescription") %>' />
<br><br>

</ItemTemplate>

<SelectedItemTemplate>

    <b>SELECTED</b><br>
    <asp:Label runat="server"
        Font-Bold="true"
        ForeColor="red"
        text='<%# DataBinder.Eval(Container.DataItem, "ProductName") %>' />
    <br>
    ProductID:
    <asp:Label runat="server"
        text='<%# DataBinder.Eval(Container.DataItem, "ProductID") %>' />
    <br>
    Manufacturer:
    <asp:Label runat="server"
        text='<%# DataBinder.Eval(Container.DataItem, "Manufacturer") %>' />
    <br>
    <asp:Label runat="server" font-italic="true"
        text='<%#
            DataBinder.Eval(Container.DataItem, "ProductDescription") %>' />
    <br><br>

</SelectedItemTemplate>

</asp:DataList>

</BODY>
</HTML>
```

Listing 11.18 Selecting Items in a DataList Control (Visual Basic.NET)

```
<% @Page Language="VB" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.IO" %>

<script language="VB" runat="server">

Sub Page_Load(Sender As Object, E As EventArgs)

    Dim ds As DataSet = new DataSet
```

Listing 11.18 continued

```
Dim fs As FileStream = new FileStream( Server.MapPath("mydata.xml"), _
 FileMode.Open, FileAccess.Read )
Dim reader As StreamReader = new StreamReader( fs )
ds.ReadXml( reader )

fs.Close()
Dim Source As DataView = new DataView( ds.Tables(0) )
MyDataList.DataSource = Source
MyDataList.SelectedIndex = 1
MyDataList.DataBind

End Sub

</script>

<HTML>
<HEAD>
    <TITLE>Basic DataList Example</TITLE>
</HEAD>
<BODY>

<asp:DataList id="MyDataList" runat="server">

    <ItemTemplate>

        <asp:Label runat="server"
            Font-Bold="true"
            text='<%# DataBinder.Eval(Container.DataItem, "ProductName") %>' />
        <br>
        ProductID:
        <asp:Label runat="server"
            text='<%# DataBinder.Eval(Container.DataItem, "ProductID") %>' />
        <br>
        Manufacturer:
        <asp:Label runat="server"
            text='<%# DataBinder.Eval(Container.DataItem, "Manufacturer") %>' />
        <br>
        <asp:Label runat="server" font-italic="true"
            text='<%#
                DataBinder.Eval(Container.DataItem, "ProductDescription") %>' />
        <br><br>

    </ItemTemplate>

    <SelectedItemTemplate>
```

Listing 11.18 continued

```
<b>SELECTED</b><br>
<asp:Label runat="server"
    Font-Bold="true"
    ForeColor="red"
    text='<%# DataBinder.Eval(Container.DataItem, "ProductName") %>' />
<br>
ProductID:
<asp:Label runat="server"
    text='<%# DataBinder.Eval(Container.DataItem, "ProductID") %>' />
<br>
Manufacturer:
<asp:Label runat="server"
    text='<%# DataBinder.Eval(Container.DataItem, "Manufacturer") %>' />
<br>
<asp:Label runat="server" font-italic="true"
    text='<%#
        DataBinder.Eval(Container.DataItem, "ProductDescription") %>' />
<br><br>

</SelectedItemTemplate>

</asp:DataList>

</BODY>
</HTML>
```

The DataGrid Control

Unlike the Repeater and DataList controls, the DataGrid control is not template driven; rather, it is table driven. It is ideal for displaying information that contains many elements or fields or many rows and columns.

The DataGrid control also provides many advanced features that make it the right choice for many programming tasks. For example, you can use the DataGrid control for selecting items, sorting, paging, and editing in place. The following subsections describe these features in detail and provide real-world examples.

The AutoGenerateColumns Property

By default, when a data source is bound to a DataGrid control, there is no need to specify the fields of the data source. The DataGrid control contains a property called AutoGenerateColumns that indicates whether the DataGrid control is to automatically generate a column for every field in the data source. By default the AutoGenerateColumns property is set to true; however, if you need to have control over the columns that are generated, you can set this property to False and specify the columns manually. (This process is described in detail later in this chapter, in the sections “Defining Columns in a DataGrid Control.”)

Using the DataGrid Control

Listings 11.19 and 11.20 show a simple example of using a DataGrid control. The example demonstrates the DataGrid control's ability to automatically generate columns. The data that is bound to the DataGrid control is coming from the Northwind database, and it is simply returning all the records and fields in an employees table.

Listing 11.19 A Basic DataGrid Control Example (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="System.Data.SqlClient" %>
<% @Import Namespace="System.Data" %>

<script language="C#" runat="server">

void Page_Load(Object Src, EventArgs E) {

    // Create Instance of Connection and Command Object
    SqlConnection cn = new
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind");
    SqlDataAdapter adapter = new SqlDataAdapter("select * from products", cn);

    DataSet myDataSet = new DataSet();
    adapter.Fill(myDataSet, "products");

    mygrid.DataSource = myDataSet.Tables["products"];
    mygrid.DataBind();

}

</script>

<HTML>
<HEAD>
    <TITLE>Basic DataGrid Example</TITLE>
</HEAD>
<BODY>

<h1>Basic DataGrid Example</h1>
<HR>

<asp:datagrid id="mygrid" runat="server" />

</BODY>
</HTML>
```

Listing 11.20 A Basic DataGrid Control Example (Visual Basic.NET)

```
<% @Page Language="VB" %>
<% @Import Namespace="System.Data.SqlClient" %>
<% @Import Namespace="System.Data" %>

<script language="VB" runat="server">

Sub Page_Load(Sender As Object, E As EventArgs)

    'Create Instance of Connection and Command Object
    Dim cn As SqlConnection = new
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind")
    Dim adapter As SqlDataAdapter = new
        SqlDataAdapter("select * from products", cn)

    Dim myDataSet As DataSet = new DataSet
    adapter.Fill(myDataSet, "products")

    mygrid.DataSource = myDataSet.Tables("products")
    mygrid.DataBind

End Sub

</script>

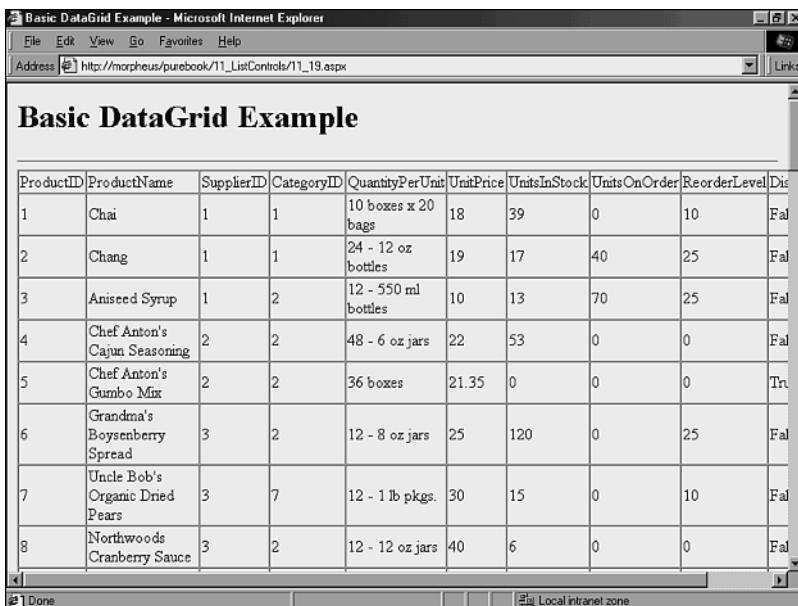
<HTML>
<HEAD>
    <TITLE>Basic DataGrid Example</TITLE>
</HEAD>
<BODY>

<h1>Basic DataGrid Example</h1>
<HR>

<asp:datagrid id="mygrid" runat="server" />

</BODY>
</HTML>
```

The definition of the `DataGridView` control is quite simple, and it actually contains only one line. When the `DataBind()` method is invoked, the `DataGridView` control automatically builds the table by creating a column for every data field and a row for every record. It even builds the header, using the field names in the data source. Figure 11.13 shows the results of Listings 11.19 and 11.20.

**Figure 11.13**

A basic DataGrid control example.

Formatting a DataGrid Control with Styles

Like the DataList control, the DataGrid control offers the ability to format a list by using styles. The properties that are available for the DataGrid control are shown in Table 11.6.

Table 11.6 Style Properties Available to the DataGrid Control

Property	Description
AlternatingItemStyle	Defines the style of all alternating items within the list
EditItemStyle	Defines the style of the item in the list that is currently being edited
FooterStyle	Defines the style of the footer in the list
HeaderStyle	Defines the style of the header in the list
ItemStyle	Defines the style of all items in the list
SelectedItemStyle	Defines the style of the item in the list that is currently selected

Each of the properties listed Table 11.6 is of the `TableItemStyle` type. Developers can set individual style attributes by using the properties of the `TableItemStyle` class, which are listed in Table 11.7.

Table 11.7 Properties of the TableItemStyle Class

Property	Description
BackColor	Sets the background color of the current item.
BorderColor	Sets the border color of the current item.
BorderStyle	Sets the border style of the current item. The following options are available: <ul style="list-style-type: none">• Dashed• Dotted• Double• Groove• Inset• None• OutSet• Ridge• Solid
BorderWidth	Sets the border width of the current item.
CssClass	Sets the <code>css</code> class of the current item.
Font-Bold	Sets the <code>Bold</code> attribute of the current item. The following options are available: <ul style="list-style-type: none">• True• False
Font-Italic	Sets the <code>Italic</code> attribute of the current item. The following options are available: <ul style="list-style-type: none">• True• False
Font-Underline	Sets the <code>Underline</code> attribute of the current item. The following options are available: <ul style="list-style-type: none">• True• False
FontSize	Sets the font size of the current item (for example, 10pt).
FontName	Sets the font of the current item (for example, Arial).
ForeColor	Sets the font color of the current item (for example, Red).
Height	Sets the height of the current item (for example, 100).
HorizontalAlign	Sets the horizontal alignment of the current item. The following options are available: <ul style="list-style-type: none">• Center• Justify• Left• Right
VerticalAlign	Sets the vertical alignment of the current item. The following options are available: <ul style="list-style-type: none">• Center• Justify• Left• Right

Table 11.7 continued

Property	Description
Width	Sets the width of the current item (for example, 100).
Wrap	Indicates whether text can wrap within the current item's cell. The following options are available: <ul style="list-style-type: none"> • True • False

The example shown in Listings 11.21 and 11.22 demonstrates how to use styles to customize the look of the DataGrid control shown in Listings 11.19 and 11.20 (see Figure 11.14).

Listing 11.21 Applying Styles to a DataGrid Control (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="System.Data.SqlClient" %>
<% @Import Namespace="System.Data" %>

<script language="C#" runat="server">

void Page_Load(Object Src, EventArgs E) {

    // Create Instance of Connection and Command Object
    SqlConnection cn = new
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind");
    SqlDataAdapter adapter = new SqlDataAdapter("select * from products", cn);

    DataSet myDataSet = new DataSet();
    adapter.Fill(myDataSet, "products");

    mygrid.DataSource = myDataSet.Tables["products"];
    mygrid.DataBind();

}

</script>

<HTML>
<HEAD>
    <TITLE>Formatting DataGrid with Styles Example</TITLE>
</HEAD>
<BODY>

<h1>Formatting DataGrid with Styles Example</h1>
<HR>

<asp:datagrid id="mygrid" runat="server"
```

Listing 11.21 continued

```
ItemStyle-Font-Name="Verdana"  
ItemStyle-Font-Size="8pt"  
AlternatingItemStyle-BackColor="Beige"  
HeaderStyle-Font-Name="Verdana"  
HeaderStyle-Font-Size="8pt"  
HeaderStyle-Font-Bold="true"  
HeaderStyle-BackColor="Maroon"  
HeaderStyle-ForeColor="White" />  
  
</BODY>  
</HTML>
```

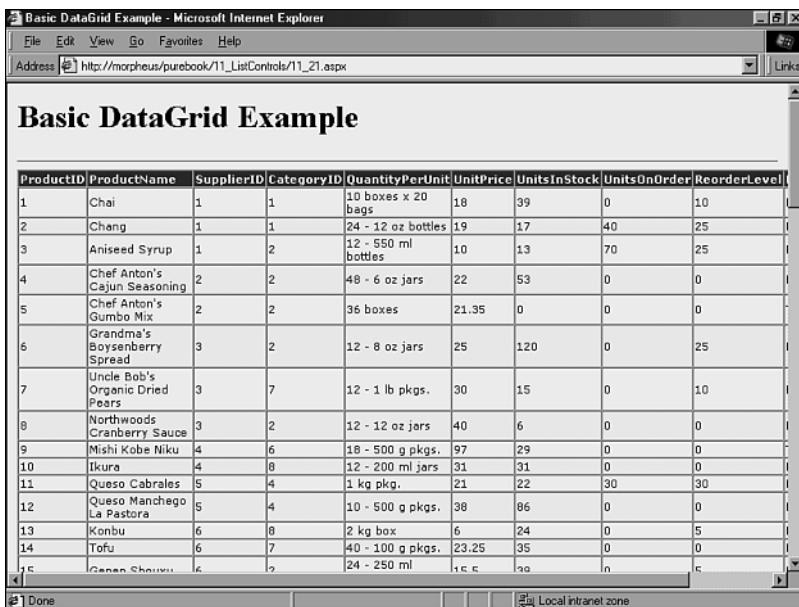
Listing 11.22 Applying Styles to a DataGrid Control (Visual Basic.NET)

```
<% @Page Language="VB" %>  
<% @Import Namespace="System.Data.SqlClient" %>  
<% @Import Namespace="System.Data" %>  
  
<script language="VB" runat="server">  
  
Sub Page_Load(Sender As Object, E As EventArgs)  
  
    'Create Instance of Connection and Command Object  
    Dim cn As SqlConnection = new  
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind")  
    Dim adapter As SqlDataAdapter = new  
        SqlDataAdapter("select * from products", cn)  
  
    Dim myDataSet As DataSet = new DataSet  
    adapter.Fill(myDataSet, "products")  
  
    mygrid.DataSource = myDataSet.Tables("products")  
    mygrid.DataBind()  
  
End Sub  
  
</script>  
  
<HTML>  
<HEAD>  
    <TITLE>Formatting DataGrid with Styles Example</TITLE>  
</HEAD>  
<BODY>  
  
<h1>Formatting DataGrid with Styles Example</h1>  
<HR>
```

Listing 11.22 continued

```
<asp:datagrid id="mygrid" runat="server"
  ItemStyle-Font-Name="Verdana"
  ItemStyle-Font-Size="8pt"
  AlternatingItemStyle-BackColor="Beige"
  HeaderStyle-Font-Name="Verdana"
  HeaderStyle-Font-Size="8pt"
  HeaderStyle-Font-Bold="true"
  HeaderStyle-BackColor="Maroon"
  HeaderStyle-ForeColor="White" />

</BODY>
</HTML>
```



The screenshot shows a Microsoft Internet Explorer window titled "Basic DataGrid Example - Microsoft Internet Explorer". The address bar shows the URL "http://morpheus/purebook/11_ListControls/11_21.aspx". The main content area displays a DataGrid control with the following data:

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel
1	Chai	1	1	10 boxes x 20 bags	18	39	0	10
2	Chang	1	1	24 - 12 oz bottles	19	17	40	25
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10	13	70	25
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22	53	0	0
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35	0	0	0
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25	120	0	25
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30	15	0	10
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40	6	0	0
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	.97	29	0	0
10	Ikura	4	8	12 - 200 ml jars	31	31	0	0
11	Queso Cabrales	5	4	1 kg pkg.	21	22	30	30
12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	.38	86	0	0
13	Konbu	6	8	2 kg box	6	24	0	5
14	Tofu	6	7	40 - 100 g pkgs.	23.25	35	0	0
15	Genen Shouyu	6	9	24 - 250 ml	15.5	70	0	0

Figure 11.14

An example of applying styles to a DataGrid control.

Defining Columns in a DataGrid Control

As mentioned previously, by default DataGrid controls automatically generate columns for each field in the bound data source. At times you need more control over what columns are created, the order in which they are created, and the types of columns that are created. To do this, you can set the AutoGenerateColumns property to False in the DataGrid control definition:

```
AutoGenerateColumns="false"
```

Next, you can define columns in the DataGrid control by creating a **Columns** section in the DataGrid control. You do this by inserting the following opening and closing tags into the DataGrid control definition:

```
<Columns>[COLUMN DEFINITIONS GO HERE]</Columns>
```

All column definitions will be inserted within these tags.

When the **AutoGenerateColumns** property is set to False, it is up to the developer to specify what columns and what type of columns are generated. The DataGrid control supports a number of different column types (see Table 11.8). The following subsections describe each of these column types in detail.

Table 11.8 Column types available to the DataGrid control

Column Type	Description
BoundColumn	A column that is bound to a specific field in the data source
TemplateColumn	A column that contains content that is rendered based on the definition of templates
HyperlinkColumn	A column that contains hyperlinks
ButtonColumn	A column that contains a number of buttons that can trigger events on a page
EditCommandColumn	A column that handles the edit commands that take place in the DataGrid control

The BoundColumn Column Type

The **BoundColumn** column type provides a way to bind a column to a field in a data source. If you simply want to display a text version of the data in a column, this is the column type you use. The **BoundColumn** type has a number of properties that enhance the flexibility of a column's content. The **BoundColumn** column type exposes the properties described in Table 11.9. The following is an example of a **BoundColumn** column type:

```
<Columns>
  <asp:BoundColumn DataField="Name" HeaderText="Author Name" />
</Columns>
```

Table 11.9 Properties of the BoundColumn Column Type

Property	Description
HeaderText	Specifies the text that appears in the column header
DataField	Indicates what field in the data source you want to bind to the column
DataFormatString	Provides the format for the data
FooterText	Specifies the text that appears in the column footer
HeaderImageUrl	Specifies the uniform resource locator (URL) for the image you want to display in the column header instead of the text specified in the HeaderText property

Table 11.9 continued

Property	Description
SortField	Indicates on what field in the data source you want the column to sort
ReadOnly	Indicates whether the column is editable if the DataGrid control is in edit mode
Visible	Indicates whether the column is visible
ItemStyle	Defines the style for the items in the column
HeaderStyle	Defines the style for the header of the column
FooterStyle	Defines the style for the footer of the column

The TemplateColumn Column Type

The `TemplateColumn` column type provides a way to control the content that is rendered in the DataGrid control's columns through the definition of templates. Templates are used within a `TemplateColumn` column type in much the same way that they are used in a Repeater or DataList control. The `TemplateColumn` column type exposes the following properties, which are described in Table 11.10. The following is an example of a `TemplateColumn` column type:

```
<Columns>
  <asp:TemplateColumn HeaderText="au_lname" SortField="au_lname">
    <ItemTemplate>
      <asp:Label runat="server"
        Text='<%# Binder.Eval(Container.DataItem, "au_lname") %>' />
    </ItemTemplate>
    <EditItemTemplate>
      <nobr>
        <asp:TextBox runat="server" id="edit_au_lname"
          Text='<%# Binder.Eval(Container.DataItem, "au_lname") %>' />
      </EditItemTemplate>
    </asp:TemplateColumn>
</Columns>
```

Table 11.10 Properties of the `TemplateColumn` Column Type

Property	Description
<code>EditItemTemplate</code>	Defines the layout for the item that is in edit mode
<code>FooterStyle</code>	Defines the style for the footer of the column
<code>FooterTemplate</code>	Defines the layout for the footer of the column
<code>FooterText</code>	Specifies the text that appears in the column footer
<code>HeaderImageURL</code>	Specifies the URL for the image you want to display in the column header instead of the text specified in the <code>HeaderText</code> property
<code>HeaderStyle</code>	Defines the style for the header of the column
<code>HeaderTemplate</code>	Defines the layout for the header of the column
<code>HeaderText</code>	Specifies the text that appears in the column header

Table 11.10 continued

Property	Description
ItemStyle	Defines the style for all the items in the column
ItemTemplate	Defines the layout for all the items in the column
SortField	Indicates on what field in the data source you want the column to sort
Visible	Indicates whether the column is visible

The HyperlinkColumn Column Type

The `HyperlinkColumn` column type provides a way to create a column that contains hyperlinks. The hyperlinks that are generated can be generated based on a bound data source, or they can be generated manually. As detailed in Table 11.11, the `HyperlinkColumn` column type has a number of properties that enhance the flexibility of the column's content. The following is an example of a `HyperlinkColumn` column type:

```
<Columns>
    <asp:HyperlinkColumn Text="View"
        NavigateUrl="view.aspx" HeaderText="View Details" />
</Columns>
```

Table 11.11 Properties of the `HyperlinkColumn` Column Type

Property	Description
DataNavigateUrlField	Specifies the field in the data source that contains the URL
DataNavigateUrlFormatString	Defines the format for the URL
DataTextField	Specifies the field in the data source that is displayed in the column
DataTextFormatString	Defines the format for the text that is displayed in the column
FooterStyle	Defines the style for the footer of the column
FooterText	Specifies the text appears in the column footer
HeaderImageUrl	Specifies the URL for the image you want to display in the column header instead of the text specified in the <code>HeaderText</code> property
HeaderStyle	Defines the style for the header of the column
HeaderText	Specifies the text that appears in the column header
ItemStyle	Defines the style for all the items in the column
NavigateUrl	Indicates the column's URL
SortField	Indicates on what field in the data source you want the column to sort
Target	Indicates the link target for the column
Text	Indicates the text that is displayed in the column
Visible	Indicates whether the column is visible

The ButtonColumn Column Type

The ButtonColumn column type provides a way to create a column that contains a number of buttons that can trigger events on the page. This type of column can be used to trigger such things as selecting an item in a DataGrid control. As detailed in Table 11.12, the ButtonColumn column type has a number of properties exposed. The following is an example of a ButtonColumn column type:

```
<Columns>
    <asp:ButtonColumn ButtonType="pushbutton"
        DataTextField="au_id" HeaderText="ButtonColumn" />
</Columns>
```

Table 11.12 Properties of the ButtonColumn Column Type

Property	Description
ButtonType	Defines the type of button to display in the column; the options are LinkButton and PushButton
CommandName	Defines the command for the column
DataTextField	Specifies the field in the data source that is displayed in the column
DataTextFormatString	Defines the format for the text that is displayed in the column
FooterStyle	Defines the style for the footer of the column
FooterText	Specifies the text that appears in the column footer
HeaderImageUrl	Specifies the URL for the image you want to display in the column header instead of the text specified in the HeaderText property
HeaderStyle	Defines the style for the header of the column
HeaderText	Specifies the text that appears in the column header
ItemStyle	Defines the style for all the items in the column
SortField	Indicates on what field in the data source you want the column to sort
Text	Indicates the text that is displayed in the column
Visible	Indicates whether the column is visible

The EditCommandColumn Column Type

The EditCommandColumn column type provides an easy way to add a column that handles the edit commands that take place in the DataGrid control. When you add an EditCommandColumn column type the DataGrid control, a column is added that contains either an Edit button or Update and Cancel buttons. The EditCommandColumn exposes the following properties, which are described in Table 11.13:

```
<Columns>
    <asp:EditCommandColumn EditText="Edit"
        CancelText="Cancel" UpdateText="OK" />
</Columns>
```

Table 11.13 Properties of the EditCommandColumn Column Type

Property	Description
ButtonType	Defines the type of button to display in the column; options are LinkButton and PushButton
CancelText	Indicates the text displayed for the Cancel button
EditText	Indicates the text displayed for the Edit button
FooterStyle	Defines the style for the column footer
FooterText	Specifies the text that appears in the column footer
HeaderImageUrl	Specifies the URL for the image you want to display in the column header instead of the text specified in the HeaderText property
HeaderStyle	Defines the style for the column header
HeaderText	The text that appears in the column header
ItemStyle	Defines the style for all the items in the column
SortField	Indicates on what field in the data source you want the column to sort
UpdateText	Indicates the text displayed for the Update button
Visible	Indicates whether the column is visible

Selecting Items in a DataGrid Control

One of the major features of the DataGrid control is that you can select items in it. By selecting items in a DataGrid control, you can use additional features, such as creating a Master/Detail view, which is covered later in this chapter.

The example shown in Listings 11.23 and 11.24 demonstrates a simple DataGrid control for which you can select items (see Figure 11.15). Each selected item's DataKey field is displayed to the user.

Listing 11.23 Selecting Items in a DataGrid Control (C#)

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<html>
<head>
    <title>Selecting An Item in the DataGrid</title>
</head>

<script runat="server">

    void Page_Load(Object sender, EventArgs e) {
        if (!Page.IsPostBack) {
            bindData();
        }
    }
}
```

Listing 11.23 continued

```
void bindData() {
    MyDataGrid.DataSource = CreateDataSource();
    MyDataGrid.DataBind();
}

void Select_Item(Object sender, EventArgs e) {
    lbl.Text = "You have selected: " +
        MyDataGrid.DataKeys[MyDataGrid.SelectedIndex].ToString();
}

ICollection CreateDataSource() {

    // Create Instance of Connection and Command Object
    SqlConnection cn = new
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind");
    SqlDataAdapter adapter = new
        SqlDataAdapter("select * from products", cn);

    DataSet myDataSet = new DataSet();
    adapter.Fill(myDataSet, "products");

    DataView dv = myDataSet.Tables["products"].DefaultView;
    return dv;
}

}

</script>

<body>

<h1>Selecting An Item in the DataGrid</h1>
<hr>

<form runat="server" id=form1 name=form1>

<asp:DataGrid
    id="MyDataGrid"
    runat="server"
    AutoGenerateColumns="false"
    DataKeyField="ProductId"
    SelectedItemStyle-Backcolor="yellow"
    HeaderStyle-BackColor="Gray"
    HeaderStyle-ForeColor="white"
    OnSelectedIndexChanged="Select_Item">

    <Columns>
```

Listing 11.23 continued

```
<asp:ButtonColumn  
    HeaderStyle-Width="100px"  
    HeaderText="Read Post"  
    Text="Select"  
    CommandName="Select"/>  
  
<asp:BoundColumn  
    HeaderStyle-Width="100px"  
    HeaderText="ID"  
    DataField="ProductId"/>  
  
<asp:BoundColumn  
    HeaderStyle-Width="200px"  
    HeaderText="Product Name"  
    DataField="ProductName"/>  
  
</Columns>  
  
</asp:DataGrid>  
  
</form>  
  
<asp:label id="lbl" runat="server"/>  
  
</body>  
</html>
```

Listing 11.24 Selecting Items in a DataGrid Control (Visual Basic.NET)

```
<%@ Page Language="VB" %>  
<%@ Import Namespace="System.Data" %>  
<%@ Import Namespace="System.Data.SqlClient" %>  
  
<html>  
<head>  
    <title>Selecting An Item in the DataGrid</title>  
</head>  
  
<script runat="server">  
  
Sub Page_Load(Sender As Object, E As EventArgs)  
    If Not Page.IsPostBack Then  
        bindData()  
    End If  
End Sub  
  
Sub bindData()  
    MyDataGrid.DataSource = CreateDataSource()
```

Listing 11.24 continued

```
MyDataGrid.DataBind()
End Sub

Sub Select_Item(Sender As Object, E As EventArgs)
    lbl.Text = "You have selected: " +
        MyDataGrid.DataKeys(MyDataGrid.SelectedIndex).ToString()
End Sub

Function CreateDataSource() As ICollection
    'Create Instance of Connection and Command Object
    Dim cn As SqlConnection = New_
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind")
    Dim adapter As SqlDataAdapter = New_
        SqlDataAdapter("select * from products", cn)

    Dim myDataSet As DataSet = new DataSet
    adapter.Fill(myDataSet, "products")

    Dim dv As DataView = myDataSet.Tables("products").DefaultView
    CreateDataSource = dv

End Function

</script>

<body>

<h1>Selecting An Item in the DataGrid</h1>
<hr>

<form runat="server" id=form1 name=form1>

<asp:DataGrid
    id="MyDataGrid"
    runat="server"
    AutoGenerateColumns="false"
    DataKeyField="ProductId"
    SelectedItemStyle-Backcolor="yellow"
    HeaderStyle-BackColor="Gray"
    HeaderStyle-ForeColor="white"
    OnSelectedIndexChanged="Select_Item">

    <Columns>

        <asp:ButtonColumn
            HeaderStyle-Width="100px"
            HeaderText="Read Post"
```

Listing 11.24 continued

```
Text="Select"
CommandName="Select"/>

<asp:BoundColumn
    HeaderStyle-Width="100px"
    HeaderText="ID"
    DataField="ProductId"/>

<asp:BoundColumn
    HeaderStyle-Width="200px"
    HeaderText="Product Name"
    DataField="ProductName"/>

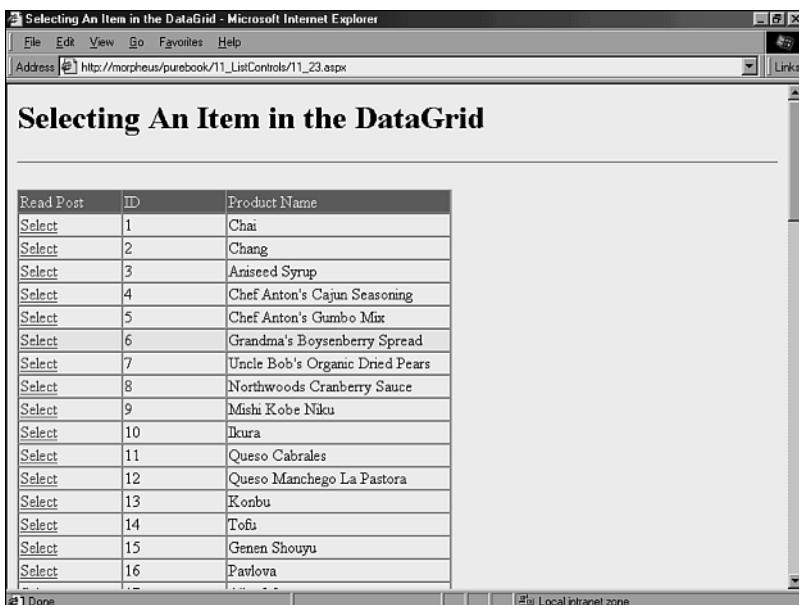
</Columns>

</asp:DataGrid>

</form>

<asp:label id="lbl" runat="server"/>

</body>
</html>
```

**Figure 11.15**

An example of selecting items in a DataGrid control.

Paging DataGrid Items

Paging allows a DataGrid control to display its items over a number of pages. This is particularly useful if the DataGrid control is bound to a data source that has a large number of records.

The AllowPaging property turns paging on and off for a DataGrid control. If the DataGrid control supports paging, this property should be set to True, as in the following line:

```
AllowPaging="true"
```

The PagerStyle property provides control of how the DataGrid control displays the paging user interface. The PagerStyle property exposes many properties, as shown in Table 11.14.

Table 11.14 PagerStyle Properties

Property	Description
Mode	Indicates whether to display number links for the different pages or whether to display simply previous/next links. The following options are available: <ul style="list-style-type: none"> • NumericPages • NextPrev
NextPageText	Indicates the text for the next page link if the mode is set to NextPrev.
PrevPageText	Indicates the text for the previous page link if the mode is set to NextPrev.
Position	Indicates where you want the paging links to appear. The following options are available: <ul style="list-style-type: none"> • Top • Bottom • TopAndBottom
PageButtonCount	Indicates how many number links are displayed if the mode is set to NumericPages.

Listing 11.25 and 11.26 demonstrate a very simple DataGrid control with paging (see Figure 11.16). Because PageSize is set to 10, each page shows 10 records.

Listing 11.25 Paging DataGrid Control Items (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="System.Data.SqlClient" %>
<% @Import Namespace="System.Data" %>

<script language="C#" runat="server">

void Page_Load(Object Src, EventArgs E) {
```

Listing 11.25 continued

```
if (!IsPostBack) BindData();  
}  
  
void BindData() {  
  
    // Create Instance of Connection and Command Object  
    SqlConnection cn = new  
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind");  
    SqlDataAdapter adapter = new SqlDataAdapter("select * from products", cn);  
  
    DataSet myDataSet = new DataSet();  
    adapter.Fill(myDataSet, "products");  
  
    mygrid.DataSource = myDataSet.Tables["products"];  
    mygrid.DataBind();  
  
}  
  
void PageData(Object src, DataGridPageChangedEventArgs e) {  
  
    mygrid.CurrentPageIndex = e.NewPageIndex;  
    BindData();  
  
}  
  
</script>  
  
<HTML>  
<HEAD>  
    <TITLE>Paging DataGrid Example</TITLE>  
</HEAD>  
<BODY>  
  
<h1>Paging DataGrid Example</h1>  
<HR>  
  
<form runat="server">  
  
    <asp:datagrid id="mygrid" runat="server"  
        PagerStyle-Mode="NextPrev"  
        PagerStyle-NextPageText="Next Page"  
        PagerStyle-PrevPageText="Previous Page"  
        PagerStyle-Visible="True"  
        AllowPaging="true"  
        PageSize="10" />
```

Listing 11.25 continued

```
OnPageIndexChanged="PageData" />

</form>

</BODY>
</HTML>
```

Listing 11.26 Paging DataGrid Control Items (Visual Basic.NET)

```
<% @Page Language="VB" %>
<% @Import Namespace="System.Data.SqlClient" %>
<% @Import Namespace="System.Data" %>

<script runat="server">

Sub Page_Load(Sender As Object, E As EventArgs)
    If Not Page.IsPostBack Then
        bindData()
    End If
End Sub

Sub BindData()

    'Create Instance of Connection and Command Object
    Dim cn As SqlConnection = New_
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind")
    Dim adapter As SqlDataAdapter = New_
        SqlDataAdapter("select * from products", cn)

    Dim myDataSet As DataSet = new DataSet
    adapter.Fill(myDataSet, "products")

    mygrid.DataSource = myDataSet.Tables("products")
    mygrid.DataBind()

End Sub

Sub PageData(src As Object, e As DataGridPageChangedEventArgs)

    mygrid.CurrentPageIndex = e.NewPageIndex
    BindData()

End Sub

</script>

<HTML>
```

Listing 11.26 continued

```
<HEAD>
    <TITLE>Paging DataGrid Example</TITLE>
</HEAD>
<BODY>

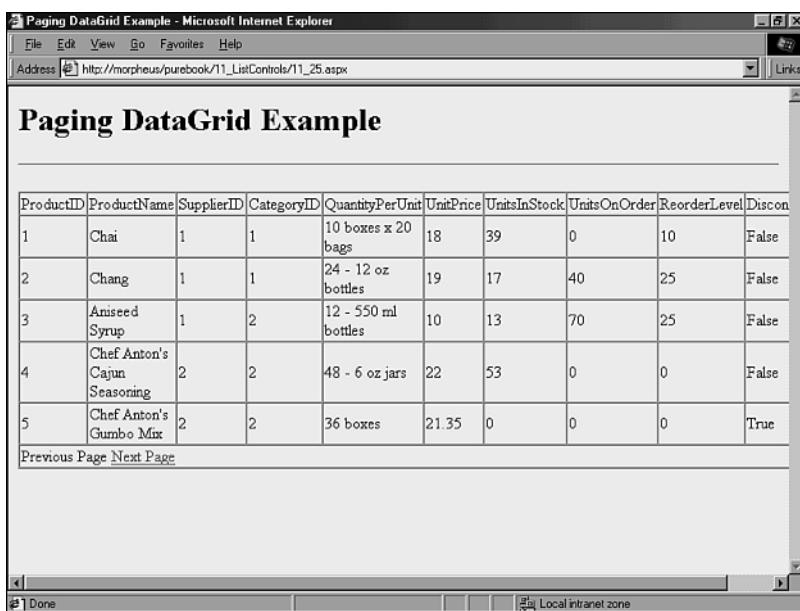
<h1>Paging DataGrid Example</h1>
<HR>

<form runat="server" id=form1 name=form1>

    <asp:datagrid id="mygrid" runat="server"
        PagerStyle-Mode="NextPrev"
        PagerStyle-NextPageText="Next Page"
        PagerStyle-PrevPageText="Previous Page"
        PagerStyle-Visible="True"
        AllowPaging="true"
        PageSize="10"
        OnPageIndexChanged="PageData" />

</form>

</BODY>
</HTML>
```

**Figure 11.16**

An example of paging items in a DataGrid control.

Sorting Items in a DataGrid Control

Sorting is another rich feature that the DataGrid control supports. Sorting adds interactivity to the read-only display of the data within a DataGrid control. Implementing sorting for a DataGrid control is so simple that huge grids of data can be modified to support sorting with only a few lines of code. When sorting is enabled for a DataGrid control, the headers for the columns that support sorting are turned into hyperlinks; when a hyperlinked column header is clicked, it sorts the data by column. Figure 11.17 shows an example of a DataGrid control that has sorting enabled.

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
17	Alice Mutton	7	6	20 - 1 kg tins	39	0	0	0	True
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10	13	70	25	False
40	Boston Crab Meat	19	8	24 - 4 oz tins	18.4	123	0	30	False
60	Camembert Pierrot	28	4	15 - 300 g rounds	34	19	0	0	False
18	Carnarvon Tigers	7	8	16 kg pkg	62.5	42	0	0	False
1	Chai	1	1	10 boxes x 20 bags	18	39	0	10	False
2	Chang	1	1	24 - 12 oz bottles	19	17	40	25	False
39	Chartreuse verte	18	1	750 cc per bottle	18	69	0	5	False
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22	53	0	0	False
f	Chef Anton's	~	~	~	~	~	~	~	True

Figure 11.17

An example of sorting items in a DataGrid control.

The DataGrid control contains a property named `AllowSorting`, which if set indicates that the DataGrid control allows sorting for any column that supports it. The first step to active sorting for the DataGrid control is to set this property to true in the DataGrid control definition.

When a column header is clicked on, the DataGrid control's `OnSortCommand` event is fired. A method is hooked to this event by setting the `OnSortCommand` property to the method's name.

Listings 11.27 and 11.28 demonstrates sorting within a DataGrid control. When you set the `AllowSorting` property to True, the column headers become links in the DataGrid control. The `OnSortCommand` property indicates that the `SortData()` method is run when the column header is clicked.

Listing 11.27 Sorting a DataGrid Control (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="System.Data.SqlClient" %>
<% @Import Namespace="System.Data" %>

<script language="C#" runat="server">

void Page_Load(Object Src, EventArgs E) {
    if (!IsPostBack) BindGrid("");
}

void BindGrid(string sortField) {

    // Create Instance of Connection and Command Object
    SqlConnection cn = new
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind");
    SqlDataAdapter adapter = new SqlDataAdapter("select * from products", cn);

    DataSet myDataSet = new DataSet();
    adapter.Fill(myDataSet, "products");

    DataView source = new DataView(myDataSet.Tables["products"]);
    source.Sort = sortField;

    mygrid.DataSource = source;
    mygrid.DataBind();
}

void SortGrid(Object src, DataGridSortCommandEventArgs e) {
    BindGrid((string)e.SortExpression);
}

</script>

<HTML>
<HEAD>
    <TITLE>Sorting DataGrid Example</TITLE>
</HEAD>
<BODY>

<h1>Sorting DataGrid Example</h1>
<HR>
```

Listing 11.27 continued

```
<form runat="server">

<asp:datagrid id="mygrid" runat="server"
    AllowSorting="true"
    OnSortCommand="SortGrid" />

</form>

</BODY>
</HTML>
```

Listing 11.28 Sorting a DataGrid Control (Visual Basic.NET)

```
<% @Page Language="VB" %>
<% @Import Namespace="System.Data.SqlClient" %>
<% @Import Namespace="System.Data" %>

<script runat="server">

Sub Page_Load(Ssrc As Object, E As EventArgs)
    If Not IsPostBack Then
        BindGrid("")
    End If
End Sub

Sub BindGrid(SortField As String)
    'Create Instance of Connection and Command Object
    Dim cn As SqlConnection = New_
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind")
    Dim adapter As SqlDataAdapter = New_
        SqlDataAdapter("select * from products", cn)

    Dim myDataSet As DataSet = new DataSet
    adapter.Fill(myDataSet, "products")

    Dim Source As DataView = myDataSet.Tables("products").DefaultView
    Source.Sort = sortField

    mygrid.DataSource = Source
    mygrid.DataBind()
End Sub

Sub SortGrid(Src As Object, E As DataGridSortCommandEventArgs)
```

Listing 11.28 continued

```
BindGrid(E.SortExpression)

End Sub

</script>

<HTML>
<HEAD>
    <TITLE>Sorting DataGrid Example</TITLE>
</HEAD>
<BODY>

<h1>Sorting DataGrid Example</h1>
<HR>

<form runat="server">

<asp:datagrid id="mygrid" runat="server"
    AllowSorting="true"
    OnSortCommand="SortGrid" />

</form>

</BODY>
</HTML>
```

Creating a Master/Detail View

Creating a Master/Detail view involves using a number of the features available in the `DataList` and `DataGrid` controls. In this example, an initial `DataGrid` control displays a collection of data, and when a user clicks an item in the `DataGrid` control, details for that item are displayed in a `DataList` control below the grid.

In the example shown in Listings 11.29 and 11.30, a `DataGrid` control is populated with the categories contained in the Northwind database. When a user selects one of the categories, all the products for that category are displayed in the `DataList` control below the `DataGrid` control. This is shown in Figure 11.18.

Listing 11.29 The Code Behind the Master/Detail View (C#)

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<script runat="server">

void Page_Load(Object sender, EventArgs e) {
```

Listing 11.29 continued

```
if (!Page.IsPostBack) BindCategories();  
  
}  
  
void BindCategories() {  
  
    // Create Instance of Connection and Command Object  
    SqlConnection cn = new  
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind");  
    SqlDataAdapter adapter = new  
        SqlDataAdapter("select * from categories", cn);  
  
    DataSet myDataSet = new DataSet();  
    adapter.Fill(myDataSet, "categories");  
  
    Categories.DataSource = new DataView(myDataSet.Tables["categories"]);  
    Categories.DataBind();  
  
}  
  
void BindProducts(Object sender, EventArgs e) {  
  
    // Determine which category was selected  
    int catId = (int)Categories.DataKeys[Categories.SelectedIndex];  
  
    // Add the Product Header  
    ProductHeader.Text = "Products for Category: " + catId.ToString();  
  
    // Create Instance of Connection and Command Object  
    SqlConnection cn = new  
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind");  
    SqlDataAdapter adapter = new SqlDataAdapter([SR]  
        "select * from products where categoryid = " + catId, cn);  
  
    DataSet myDataSet = new DataSet();  
    adapter.Fill(myDataSet, "products");  
  
    Products.DataSource = new DataView(myDataSet.Tables["products"]);  
    Products.DataBind();  
  
}  
  
</script>  
  
<html>  
<head>  
    <title>Master/Detail with the DataGrid and DataList</title>
```

Listing 11.29 continued

```
</head>

<body>

<h1>Master/Detail with the DataGrid and DataList</h1>
<hr>

<form runat="server" id=form1 name=form1>

    <asp:DataGrid runat="server" id="Categories"
        BorderColor="black"
        BorderWidth="1"
        GridLines="Both"
        CellPadding="3"
        CellSpacing="0"
        Font-Name="Verdana"
        Font-Size="8pt"
        AutoGenerateColumns="false"
        HeaderStyle-BackColor="silver"
        HeaderStyle-Font-Bold="true"
        DataKeyField="CategoryID"
        OnSelectedIndexChanged="BindProducts">

        <Columns>

            <asp:ButtonColumn
                HeaderStyle-Width="100px"
                HeaderText="View Products"
                Text="Select"
                CommandName="Select"/>

            <asp:BoundColumn
                HeaderStyle-Width="100px"
                HeaderText="ID"
                DataField="CategoryId"/>

            <asp:BoundColumn
                HeaderStyle-Width="200px"
                HeaderText="Category Name"
                DataField="CategoryName"/>

        </Columns>

    </asp:DataGrid>

    <br>
    <b><asp:Label id="ProductHeader" runat="server" /></b>
```

Listing 11.29 continued

```

<br><br>

<asp:DataList id="Products" runat="server"
    BorderColor="black"
    CellPadding="3"
    BorderWidth="1"
    GridLines="Both"
    RepeatColumns="2"
    RepeatDirection="Horizontal"
    ItemStyle-Width="50%"
    Font-Name="Verdana"
    Font-Size="8pt"
    HeaderStyle-BackColor="#aaaadd">

    <ItemTemplate>

        <b>Product Name:</b>
        <%# DataBinder.Eval(Container.DataItem, "ProductName") %>
        <br>
        <b>ProductID:</b>
        <%# DataBinder.Eval(Container.DataItem, "ProductId") %>
        <br>
        <b>Units In Stock:</b>
        <%# DataBinder.Eval(Container.DataItem, "UnitsInStock") %>
        <br>
        <b>Unit Price:</b>
        <%# DataBinder.Eval(Container.DataItem, "UnitPrice", "{0:c}") %>
        <br>

    </ItemTemplate>

</asp:DataList>

</form>

</body>
</html>

```

Listing 11.30 The Code Behind the Master/Detail View (Visual Basic.NET)

```

<%@ Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<script runat="server">

```

Listing 11.30 continued

```
Sub Page_Load(Sender As Object, E As EventArgs)
    If Not Page.IsPostBack Then
        BindCategories()
    End If
End Sub

Sub BindCategories()
    'Create Instance of Connection and Command Object
    Dim cn As SqlConnection = New_
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind")
    Dim adapter As SqlDataAdapter = New_
        SqlDataAdapter("select * from categories", cn)

    Dim myDataSet As DataSet = new DataSet
    adapter.Fill(myDataSet, "categories")

    Categories.DataSource = myDataSet.Tables("categories").DefaultView
    Categories.DataBind()

End Sub

Sub BindProducts(Sender As Object, E As EventArgs)
    ' Determine which category was selected
    Dim catId As Integer = Categories.DataKeys(Categories.SelectedIndex)

    ' Add the Product Header
    ProductHeader.Text = "Products for Category: " & catId.ToString()

    'Create Instance of Connection and Command Object
    Dim cn As SqlConnection = New_
        SqlConnection("server=localhost;uid=sa;pwd=;database=northwind")
    Dim adapter As SqlDataAdapter = New_
        SqlDataAdapter("select * from products where categoryid = " & catId,
cn)

    Dim myDataSet As DataSet = new DataSet
    adapter.Fill(myDataSet, "products")

    Products.DataSource = myDataSet.Tables("products").DefaultView
    Products.DataBind()

End Sub
```

Listing 11.30 continued

```
</script>

<html>
<head>
    <title>Master/Detail with the DataGrid and DataList</title>
</head>

<body>

<h1>Master/Detail with the DataGrid and DataList</h1>
<hr>

<form runat="server" id=form1 name=form1>

<asp:DataGrid runat="server" id="Categories"
    BorderColor="black"
    BorderWidth="1"
    GridLines="Both"
    CellPadding="3"
    CellSpacing="0"
    Font-Name="Verdana"
    Font-Size="8pt"
    AutoGenerateColumns="false"
    HeaderStyle-BackColor="silver"
    HeaderStyle-Bold="true"
    DataKeyField="CategoryID"
    OnSelectedIndexChanged="BindProducts">

<Columns>

    <asp:ButtonColumn
        HeaderStyle-Width="100px"
        HeaderText="View Products"
        Text="Select"
        CommandName="Select"/>

    <asp:BoundColumn
        HeaderStyle-Width="100px"
        HeaderText="ID"
        DataField="CategoryId" />

    <asp:BoundColumn
        HeaderStyle-Width="200px"
        HeaderText="Category Name"
        DataField="CategoryName" />

</Columns>
```

Listing 11.30 continued

```
</asp:DataGrid>

<br>
<b><asp:Label id="ProductHeader" runat="server" /></b>
<br><br>

<asp:DataList id="Products" runat="server"
    BorderColor="black"
    CellPadding="3"
    BorderWidth="1"
    GridLines="Both"
    RepeatColumns="2"
    RepeatDirection="Horizontal"
    ItemStyle-Width="50%"
    Font-Name="Verdana"
    Font-Size="8pt"
    HeaderStyle-BackColor="#aaaadd">

    <ItemTemplate>

        <b>Product Name:</b>
        <%# DataBinder.Eval(Container.DataItem, "ProductName") %>
        <br>
        <b>ProductID:</b>
        <%# DataBinder.Eval(Container.DataItem, "ProductId") %>
        <br>
        <b>Units In Stock:</b>
        <%# DataBinder.Eval(Container.DataItem, "UnitsInStock") %>
        <br>
        <b>Unit Price:</b>
        <%# DataBinder.Eval(Container.DataItem, "UnitPrice", "{0:c}") %>
        <br>

    </ItemTemplate>

</asp:DataList>

</form>

</body>
</html>
```

The screenshot shows a Microsoft Internet Explorer window with the title "Master/Detail with the DataGrid and DataList - Microsoft Internet Explorer". The address bar shows the URL "http://morpheus/purebook/11_ListControls/11_29.aspx". The main content area displays a "Master/Detail with the DataGrid and DataList" example.

Master Grid:

View Products	ID	Category Name
Select	1	Beverages
Select	2	Condiments
Select	3	Confections
Select	4	Dairy Products
Select	5	Grains/Cereals
Select	6	Meat/Poultry
Select	7	Produce
Select	8	Seafood

Detail View:

Products for Category: 4

Product Name: Queso Cabrales ProductID: 11 Units In Stock: 22 Unit Price: \$21.00	Product Name: Queso Manchego La Pastora ProductID: 12 Units In Stock: 86 Unit Price: \$38.00
Product Name: Gorgonzola Telino ProductID: 31 Units In Stock: 0 Unit Price: \$12.50	Product Name: Mascarpone Fabioli ProductID: 32 Units In Stock: 9 Unit Price: \$32.00
Product Name: Geitost ProductID: 33	Product Name: Raclette Courdavault ProductID: 59

Figure 11.18

An example of a Master/Detail view with DataGrid and DataList controls.

CHAPTER 12

Working with ASP.NET Validation Controls

Validation controls are used to validate the contents of input Web form controls. There are five types of validation controls built into ASP.NET—RequiredFieldValidator, RangeValidator, CompareValidator, RegularExpressionValidator, and CustomValidator—and each serves a unique validation purpose. Each of these controls is covered in depth in this chapter.

Properties and Methods Common to All Controls

Because the validation controls are all derived from the same base class, they share a common set of properties and methods, which are listed in Table 12.1.

Table 12.1 Validation Controls at a Glance

Properties	Boolean IsValid String ControlToValidate String ErrorMessage ValidationDisplay Display String Text
Method	Validate()

`IsValid` is a Boolean property that holds the current validation state of the control. When the value is set to `True`, this indicates that the control being validated has passed validation. If `IsValid` is `False`, the control has failed the validation criteria.

`ErrorMessage` is another property in the `IValidator` interface. `ErrorMessage` is a string that contains the error message that will be displayed to the user if validation fails.

ControlToValidate contains the id of the Web control on which you are performing validation. For instance, consider this TextBox Web control:

```
<asp:TextBox id="FirstName" runat="server"></asp:TextBox>
```

To make sure a user places some sort of input into this field, you could use a RequiredFieldValidator control and make sure that the ControlToValidate property of the validation control contains the ID of the control being validated:

```
<asp:RequiredFieldValidator ControlToValidate="FirstName" runat="server" />
```

The Display property controls how the validation control results appear on the page and can contain one of three values. If the Display property is set to None, the validator is invisible to the user; even when validation fails, the error message is not displayed. If the Display property is set to Static, the validator takes up a certain amount of space on the page, even when it is hidden. This is useful if the placement of the validation control would cause a shift in the appearance of HTML when the error message is displayed to the user. The final possible value for Display is Dynamic. This is precisely the opposite of Static; the control will take up space only when the error message is displayed to the user.

The Validate() method is not of much concern unless you are designing your own validation control. The framework calls Validate() whenever the control needs to perform validation. This generally happens during postback of the Web form. However, Validate() can be called whenever the page changes and the change affects the state of validation.

Placing a Control on a Web Form

How a validation control is placed on a Web form is very important to the overall effectiveness of the control. The user must be able to see the error message when validation fails.

For this reason, you should generally not place validation controls at the bottom of the page. Users with small screen sizes or users who simply do not have their browsers maximized will not be able to see the error message. A good location for a validation control is right at the top of the page, where it is sure to grab the user's attention. Another good place for a validator is directly below or next to the control it is validating. This makes the invalid input obvious quickly.

TIP

For a validation control to function in an ASP.NET page, it must be within a server-side form (for example, the validation control must be within `<form runat="server">` and `</form>` tags).

Formatting Error Messages

Because validation controls are derived from the same base class as all Web controls, the developer has access to all the same style formatting properties as for the Web

controls. For example, Font-Name and Font-Size are both formatting properties that can be specified with validation controls.

TIP

It is possible to insert HTML directly in the `ErrorMessage` property of the validation control. It is good coding practice to attempt to achieve formatting using the exposed style interfaces. However, when all else fails, the ability to insert HTML inline gives the developer almost complete control over the appearance of the error message.

The RequiredField Validator

Table 12.2 The RequiredField Validator at a Glance

Property	Type	String	InitialValue
----------	------	--------	--------------

The `RequiredField` control is the simplest of all validation controls. Applying the `RequiredField` validator to an input control makes that field required. This control is used in most instances to ensure that a field is not left blank because `InitialValue` is an empty string by default. However, if `InitialValue` is specified, then changing the targeted input control's value to an empty string will enable the control to pass validation.

Listing 12.1 demonstrates the use of three separate `RequiredField` validators on three separate input controls. The validation controls are placed in a table cell directly to the right of the input control they are validating. Therefore, when validation fails, the error message (in this case, just a simple *) is placed next to the proper control.

Listing 12.1 An Example of the RequiredField Control

```
<% @Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

<script language="C#" runat="server" >
    void Page_Load(Object Source, EventArgs E)
    {
        if(Page.IsPostBack==true)
            if(Page.IsValid==false)
                msg.Text = "All form elements are required. Please enter
                           information in fields marked with a '<font color=red>*</font>'";
            else
                msg.Text = "You have successfully entered all information!";
    }
</script>
```

Listing 12.1 continued

```
</HEAD>
<BODY>

<h2>Required Field Validator Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
<table>
<tr><td>
    Name: </td>
<td>

    <asp:TextBox id=name1 MaxLength="10" Columns="20" runat=server/></td>

<td>

    <asp:RequiredFieldValidator
        ControlToValidate="name1"
        Display="Static"
        InitialValue="" Width="100%" runat=server>
        *
    </asp:RequiredFieldValidator>

</td></tr>
<tr><td>
    Address:
</td>
<td>

    <asp:TextBox id=address1 MaxLength="40" Columns="20" runat=server/></td>

<td>

    <asp:RequiredFieldValidator
        ControlToValidate="address1"
        Display="Static"
        InitialValue="" Width="100%" runat=server>
        *
    </asp:RequiredFieldValidator>

</td></tr>
<tr><td>
    Phone Number:
</td>
<td>

    <asp:TextBox
```

Listing 12.1 continued

```
id=phone1
MaxLength="10"
Columns="20"
runat=server>
XXX-XXX-XXXX
</asp:Textbox></td>

<td>

<asp:RequiredFieldValidator
    ControlToValidate="phone1"
    Display="Dynamic"
    InitialValue="XXX-XXX-XXXX" Width="100%" runat=server>
    *
</asp:RequiredFieldValidator>

<asp:RequiredFieldValidator
    ControlToValidate="phone1"
    Display="Dynamic"
    InitialValue="" Width="100%" runat=server>
    *
</asp:RequiredFieldValidator>

</td></tr>
<tr><td>
    &nbsp;
</td>
<td>

    <asp:Button id=button1 Text="Submit" runat=server/></td></tr>

</table><br><br>

<asp:Label id=msg runat=server/>

</form>
<hr>

</BODY>
</HTML>
```

Note the use of the `Page.IsValid` property. This value is True only when all validation occurs successfully on a page, and it is False if any validation fails. Figures 12.1 and 12.2 show how the page will appear if validation fails. Note the appearance of the error message at the bottom of the page as well as the asterisks next to the required fields.

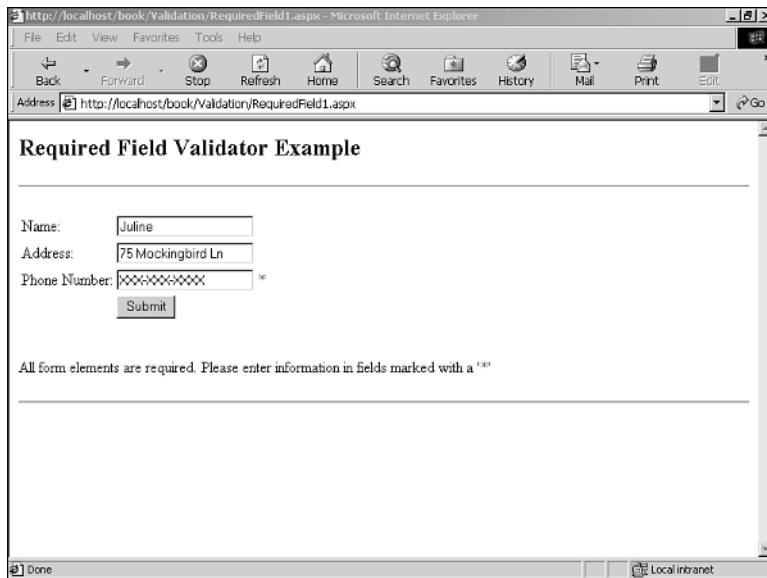


Figure 12.1

The appearance of validation controls in Listing 12.1 when validation fails for a single field.

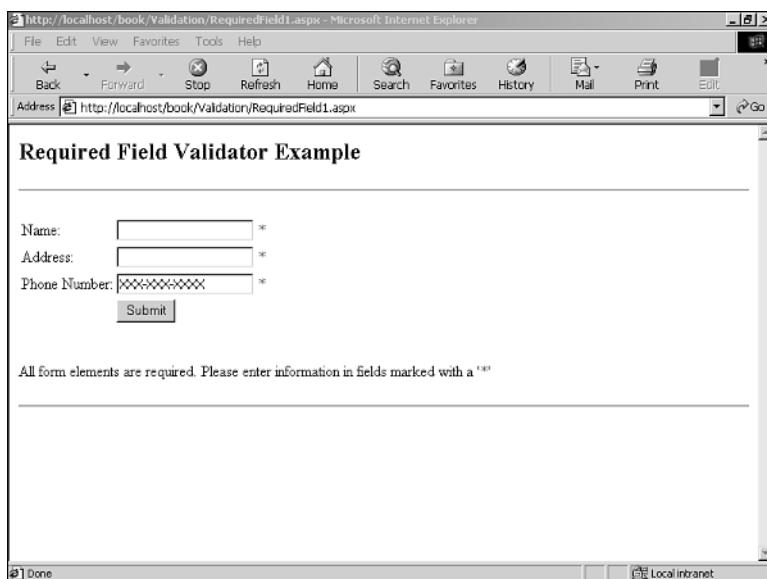


Figure 12.2

The appearance of validation controls in Listing 12.1 when multiple fields fail.

The Range Validator

Table 12.3 The Range Validator at a Glance

Properties	ValidationDataType Type
	String MinimumControl
	String MaximumControl
	String MinimumValue
	String MaximumValue

The Range validation control ensures that the value of an input control is between two values (a minimum and a maximum). The control maximum and minimum values are read from the `MinimumValue` and `MaximumValue` properties of the control. Alternatively, the control can use the value of another control as the maximum and minimum values. If used this way, the controls for maximum and minimum are specified by the `MinimumControl` and `MaximumControl` properties. If both control and constant values are specified, the control value overrides the constant value.

TIP

If the field that the Range validation control is validating is left blank, validation will succeed. This functionality was created intentionally, with the idea that if developers didn't want to allow blank values, they would use the `RequiredField` validator in addition to the Range validator.

Listing 12.2 shows how to use a Range validator to ensure that a user enters a value between 1 and 10 in a text box.

Listing 12.2 An Example of the Range Control

```
<% @Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->
</HEAD>
<BODY>

<h2>Range Validator Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
<table>
    <tr><td>
```

Listing 12.2 continued

```
Please enter a value between 1 and 10: </td>
<td>

<asp:TextBox
    id=textBox1
    MaxLength="10"
    Columns="20"
    runat=server/>
</td></tr>

<tr><td>
    &nbsp;
</td>
<td>

    <asp:Button id=button1 Text="Submit" runat=server/></td></tr>

</table><br><br>

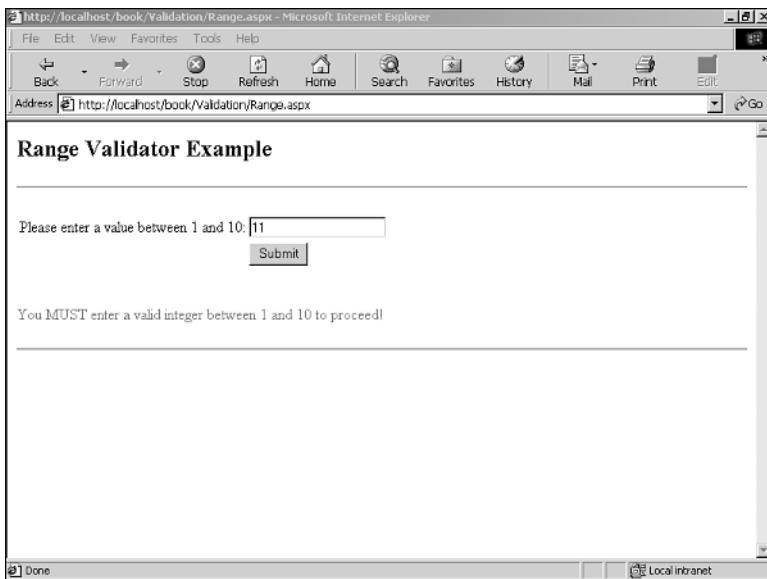
<asp:RangeValidator runat="server"
    Type="Integer"
    ErrorMessage="You MUST enter a valid integer between 1 and 10 to proceed!"
    ControlToValidate="textBox1"
    MaximumValue="10"
    MinimumValue="1">
</asp:RangeValidator>

</form>

<hr>

</BODY>
</HTML>
```

Figure 12.3 shows how the code in Listing 12.2 appears if validation fails. Notice the error message that appears at the bottom of the page. This is pulled from the ErrorMessage property of the Range validation control.

**Figure 12.3**

The appearance of the Range control in Listing 12.2 when validation fails.

The Compare Validator

Table 12.4 The Compare Validator at a Glance

Properties	ValidationDataType Type ValidationCompareOperator Operator String ControlToCompare String ValueToCompare
------------	---

Table 12.5 ValidationCompareOperator Enumeration at a Glance

Values	Equal NotEqual GreaterThan GreaterThanOrEqual LessThan LessThanOrEqual DataTypeCheck
--------	--

The Compare validator compares the value of the targeted control to a constant value or the value of another control. You specify the `ControlToCompare` property when performing a comparison to another input control value, and you use `ValueToCompare` when performing a comparison to a constant value.

The Operator property controls the type of comparison being performed. The possible values of the Operator property are shown in Table 12.5. Most are fairly self-explanatory, except for the last item, `DataTypeCheck`. If the Operator property is set to `DataTypeCheck`, then the `Compare` validator will only check the data type of the targeted control. Legitimate values for the Type field are `String`, `Integer`, `Double`, `DateTime`, and `Currency`.

Listing 12.3 shows how to use the `Compare` validation control to perform a very common action in Web applications: Comparing two password fields to ensure that they match.

Listing 12.3 An Example of a Compare Control

```
<% @Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Sender, EventArgs e)
        {
            if(Page.IsPostBack==true)
                if(Page.IsValid==true)
                    msg.Text="Your input has been successfully received!";
                else
                    msg.Text="";
        }
    </script>

</HEAD>
<BODY>

<h2>Compare Validator Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <table>
        <tr><td>
            Password: </td>
        <td>
            <asp:TextBox
                id=password1
                MaxLength="10"
                TextMode="Password"
                Columns="20"
                runat=server/>
        </td>
    </tr>
</table>
</form>
```

Listing 12.3 continued

```
</td></tr>

<tr><td>
    Confirm Password:
</td>
<td>

    <asp:TextBox
        id=password2
        MaxLength="10"
        TextMode="Password"
        Columns="20"
        runat=server/>
</td></tr>

<tr><td>
    &nbsp;
</td>
<td>

    <asp:Button id=button1 Text="Submit" runat=server/></td></tr>

</table><br><br>

<asp:CompareValidator
    ControlToValidate="password2"
    ControlToCompare="password1"
    Display="dynamic"
    Font-Size="10pt"
    ErrorMessage="The password fields do not match. Please re-enter."
    Type="String"
    runat=server/>

<asp:RequiredFieldValidator
    ControlToValidate="password1"
    Display="dynamic"
    Font-Name="verdana"
    Font-Size="10pt"
    ErrorMessage="'Password' must not be left blank."
    runat=server/>

<asp:Label id=msg runat=server/>

</form>
<hr>

</BODY>
</HTML>
```

Figure 12.4 shows how the code in Listing 12.3 appears when validation fails. The error message that is displayed at the bottom of the page is pulled from the ErrorMessage property of CompareValidator. If the password field had been left blank, the ErrorMessage of the RequiredFieldValidator would have been displayed as well.

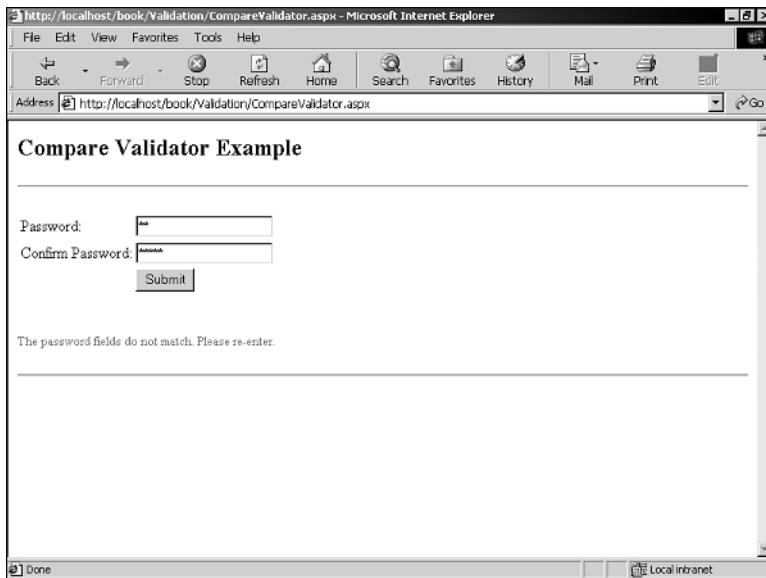


Figure 12.4

The appearance of the Compare control in Listing 12.3 when validation fails.

The RegularExpression Validator

Table 12.6 The RegularExpression Validator at a Glance

Properties	String ValidationExpression
------------	-----------------------------

The RegularExpression validation control performs validation on a targeted field, using a special shorthand language called regular expressions. Regular expressions allow you to filter, sort through, and manipulate text easily. This makes it easy to perform validation to make sure that the user has entered a valid Social Security number or phone number, for example.

Listing 12.4 shows how to use a RegularExpression validation control to ensure that a user has entered a correct e-mail address.

Listing 12.4 An Example of the RegularExpression Control

```
<% @Page Language="C#" %>
```

```
<HTML>
```

Listing 12.4 continued

```
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {
            if(Page.IsPostBack==true)
                if(Page.IsValid==true)
                    msg.Text = "You have successfully entered all information!";
        }
    </script>

</HEAD>
<BODY>

<h2>Regular Expression Validator Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
<table>
<tr><td>
    Please enter your email address: </td>
<td>

    <asp:TextBox id=name1 MaxLength="100" Columns="20" runat=server/></td>

<td>

    <asp:RegularExpressionValidator
        ControlToValidate="name1"
        Display="Static"
        ValidationExpression="^[\w-]+@[ \w-]+\.(com|net|org|edu|mil)$"
        Font-Name="Arial" Font-Size="11"
        runat=server>
        Not a valid e-mail address
    </asp:RegularExpressionValidator>

    <asp:RequiredFieldValidator
        ControlToValidate="name1"
        Display="Static"
        InitialValue="" Width="100%" runat=server>
        *
    </asp:RequiredFieldValidator>

</td></tr>
<tr><td>
```

Listing 12.4 continued

```
&nbsp;
</td>
<td>

<asp:Button id=button1 Text="Submit" runat=server/></td></tr>

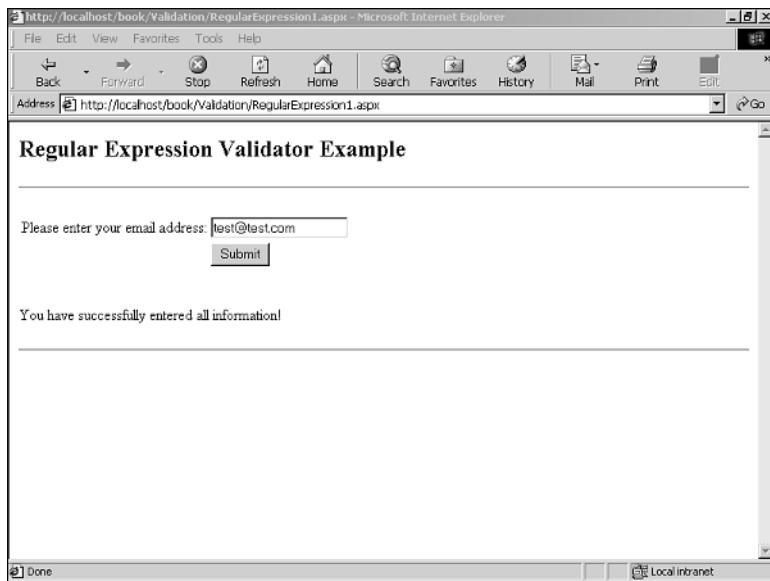
</table><br><br>

<asp:Label id=msg Text="" runat=server/>

</form>
<hr>

</BODY>
</HTML>
```

In the example shown in Listing 12.4, if the user enters a valid e-mail address (according to the restrictions in Listing 12.4), he or she sees a screen like the one shown in Figure 12.5.

**Figure 12.5**

The appearance of the RegularExpression controls in Listing 12.4 upon successful validation.

However, if the user enters an invalid e-mail address or a random string, the validation controls display errors, as in Figure 12.6.

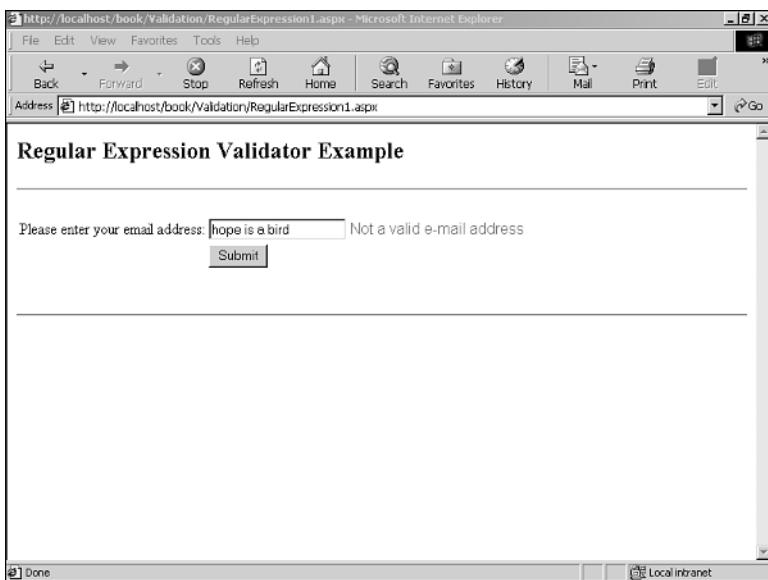


Figure 12.6

The appearance of the RegularExpression controls in Listing 12.4 when validation fails.

Users who are familiar with Unix operating systems might be familiar with regular expression syntax because it is used in commands such as grep. Microsoft .NET's implementation of regular expressions derives from Perl. A full discussion of regular expressions is beyond the scope of this book. However, msdn.microsoft.com/library/dotnet/cpguide/cpconcomregularexpressions.htm provides more information on and examples of regular expressions.

The Custom Validator

Table 12.7 The Custom Validator at a Glance

Properties	String ClientValidationFunction Boolean OnServerValidate
------------	---

The built-in validation controls perform most of the common forms of validation that a typical Web application would provide. However, there are certain situations that require much more granular control of the validation process such as credit card validation.

In these special situations, the `Custom` validation control is useful. The `ClientValidationFunction` property of the `Custom` validator allows the developer to specify a client-side function that will be used to perform validation logic. This function returns `True` upon a successful validation and `False` if the validation fails. This client-side validation performs no round-trip to the server.

Similarly, the `ServerValidationFunction` property enables the developer to specify a server-side function to perform validation. The function returns True if validation succeeds and False if validation fails. If both the `ServerValidationFunction` and `ClientValidationFunction` properties are specified, validation occurs on both the client and server. However, if the server-side validation fails, the validation fails, even if the client-side validation succeeds.

Listing 12.5 contains a `Custom` validation control, which is used to make sure the input is a prime number between 1 and 20. In addition to the `Custom` validator, a `RequiredField` validator and a `Range` validator are also used.

Listing 12.5 An Example of a Custom Control

```
<% @Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >

        bool ServerValidate (object source, string value)
        {
            try {
                int i = Int32.Parse(value);

                if ( (i % 2 == 0)|| (i % 3 == 0) && (i!=2) && (i!=3) )
                    if( i!=2 )
                    {
                        msg.Text = "Please enter a prime number between 1 and 20!";
                        return false;
                    }
            }
            catch (Exception) {}

            msg.Text = value + " is a prime number!";
            return true;
        }
    </script>

</HEAD>
<BODY>
```

Listing 12.5 continued

```
<h2>Custom Validation Example</h2>
<hr>

<form runat="server">

    <asp:Label
        id=msg
        Text="Please enter a prime number between 1 and 20:"
        runat=server/><br>

    <br><br>

    <asp:TextBox id=text1 runat="server" />

    <asp:RequiredFieldValidator id="RequiredFieldValidator1" runat="server"
        ControlToValidate="text1"
        ErrorMessage="Field cannot be blank!"
        Display="Dynamic"
        Font-Name="verdana" Font-Size="10pt">
    </asp:RequiredFieldValidator>

    <asp:CustomValidator id="CustomValidator1" runat="server"
        ControlToValidate="text1"
        ErrorMessage="Not a prime number!"
        ClientValidationFunction="ClientValidate"
        OnServerValidate="ServerValidate"
        Display="Static"
        Font-Name="verdana" Font-Size="10pt">
    </asp:CustomValidator>

    <asp:RangeValidator
        Type="Integer"
        ErrorMessage="You must enter a valid integer between 1 and 20!"
        Font-Name="verdana" Font-Size="10pt"
        ControlToValidate="text1"
        MaximumValue="20"
        MinimumValue="1"
        runat="server">
    </asp:RangeValidator>

    <br><br>

    <asp:Button id=button1 text="Submit" runat="server" />
```

Listing 12.5 continued

```
<script language="javascript">

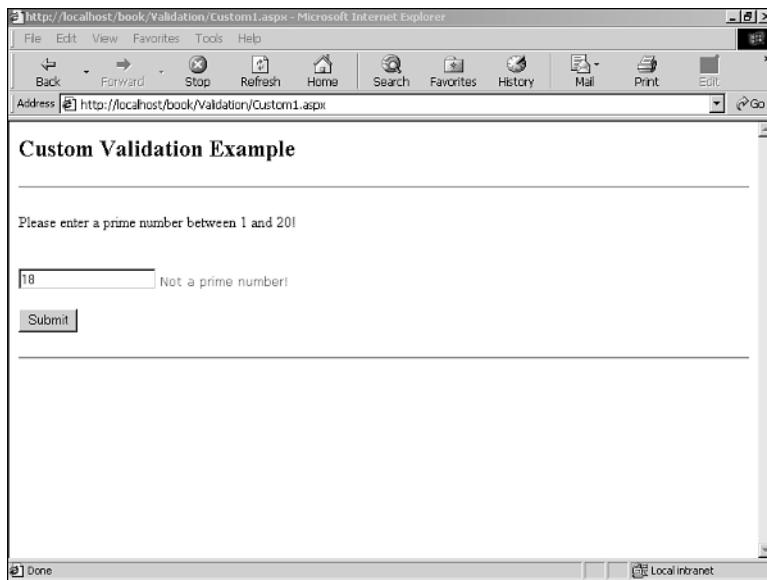
    function ClientValidate(source, value)
    {
        if ( (value % 2 == 0)|| (value % 3 == 0) && (value!=2) && (value!=3) )
            return false;
        else
            return true;
    }

</script>

</form>
<hr>

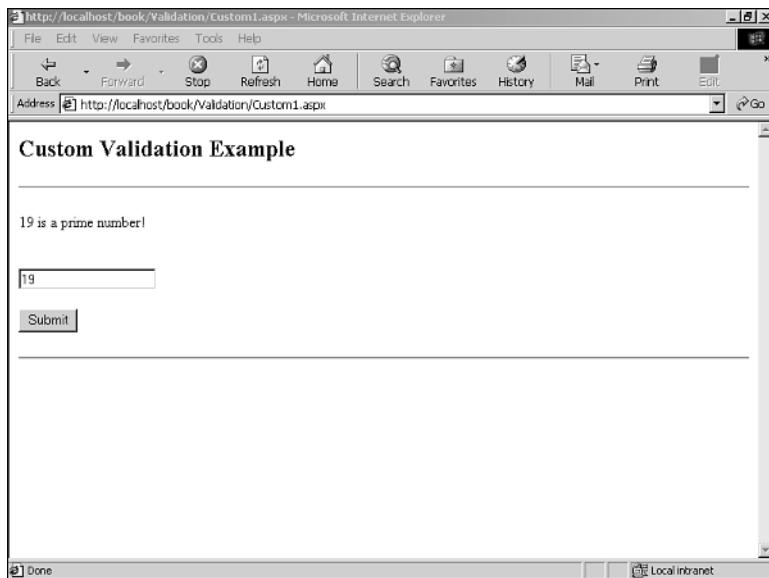
</BODY>
</HTML>
```

In this example, if the user does not enter a prime number between 1 and 20, a screen like the one in Figure 12.7 appears.

**Figure 12.7**

The appearance of the custom validator when an invalid input is entered.

When the user enters a valid prime number between 1 and 20, the screen changes, as shown in Figure 12.8.

**Figure 12.8**

The appearance of the custom validator when an invalid input is changed to a correct value.

The ValidationSummary Web Control

Table 12.8 The ValidationSummary Web Control at a Glance

Properties	String HeaderText
	ValidationSummaryDisplayMode DisplayMode
	Boolean ShowSummary
	Boolean ShowMessageBox

The ValidationSummary Web control enables the developer to list all the validation errors on a page in one grouping instead of having to loop through all the validation controls on an ASP.NET page and manually check to see the validation state of the control. This can be used to easily show all validation errors at the top of the page instead of listing them in each validation control.

The HeaderText property enables the developer to specify a header for the listing that is displayed only when validation fails. When the ShowSummary property is set to False, the ValidationSummary control does not display the validation error information inline. The ShowMessageBox property enables a developer to display a message box to users with uplevel browsers. An *uplevel browser* is any Microsoft Internet Explorer browser version 4 and later or Netscape Navigator version 6 or later.

TIP

The `Text` property of a Validation control can be set instead of the `ErrorMessage` property to display an error message. However, in the Validation Summary, only the string in the `ErrorMessage` property of the validation control is displayed.

The `DisplayMode` property controls how the `ValidationSummary` control displays the error information. The three possible values for this property are `List`, `BulletList`, and `SingleParagraph`.

Listing 12.6 uses the `ValidationSummary` control to display error information to the client instead of listing errors individually. Note that the validation controls themselves are hidden.

Listing 12.6 An Example of the ValidationSummary Control

```
<% @Page Language="C#" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {
            if(Page.IsPostBack==true)
                if(Page.IsValid==true)
                    msg.Text = "You have successfully entered all information!";
                else
                    msg.Text = "";
        }
    </script>

</HEAD>
<BODY>

<h2>Validation Summary Example</h2>
<hr>

<form runat="server" id=form1 name=form1>
    <table>
        <tr><td>

            <asp:ValidationSummary runat="server"
                HeaderText="Please check your entries in the following fields:"
                DisplayMode="BulletList"
            >
        </td>
    </tr>
</table>
</form>

```

Listing 12.6 continued

```
Font-Name="verdana"
Font-Size="12">
</asp:ValidationSummary>

</td></tr>
<tr><td>
    Name: </td>
<td>

    <asp:TextBox id=name1 MaxLength="10" Columns="20" runat=server/></td>

<td>

    <asp:RequiredFieldValidator
        ControlToValidate="name1"
        ErrorMessage="'Name' field cannot be left blank"
        Display="Dynamic"
        InitialValue="" Width="100%" runat=server>
    *
    </asp:RequiredFieldValidator>

</td></tr>
<tr><td>
    Address:
</td>
<td>

    <asp:TextBox id=address1 MaxLength="40" Columns="20" runat=server/></td>

<td>

    <asp:RequiredFieldValidator
        ControlToValidate="address1"
        ErrorMessage="'Address' field cannot be left blank"
        Display="Dynamic"
        InitialValue="" Width="100%" runat=server>
    *
    </asp:RequiredFieldValidator>

</td></tr>
<tr><td>
    Phone Number:
</td>
<td>

    <asp:TextBox id=phone1 MaxLength="10" Columns="20" runat=server>
```

Listing 12.6 continued

```
XXX-XXX-XXXX
</asp:Textbox></td>

<td>

<asp:RequiredFieldValidator
    ControlToValidate="phone1"
    ErrorMessage="You must enter a valid phone number!"
    Display="Dynamic"
    InitialValue="XXX-XXX-XXXX" Width="100%" runat=server>
    *
</asp:RequiredFieldValidator>

<asp:RequiredFieldValidator
    ControlToValidate="phone1"
    ErrorMessage="You must enter a valid phone number!"
    Display="Dynamic"
    InitialValue="" Width="100%" runat=server>
    *
</asp:RequiredFieldValidator>

</td></tr>
<tr><td>
    &nbsp;
</td>
<td>

    <asp:Button id=button1 Text="Submit" runat=server/></td></tr>

</table><br><br>
</form>

    <asp:Label id=msg Text="" runat=server/>
<hr>

</BODY>
</HTML>
```

The **ValidationSummary** control displays the error information from each control in one place on the page, whether it's the errors from a single control, as in Figure 12.9, or the errors from all controls on the page, as in Figure 12.10.

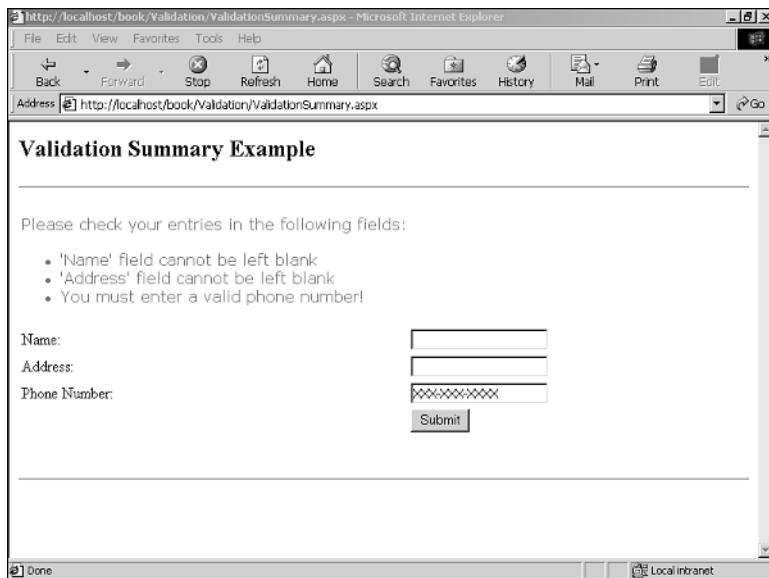


Figure 12.9

The appearance of validationSummary control output with an error from a single control.

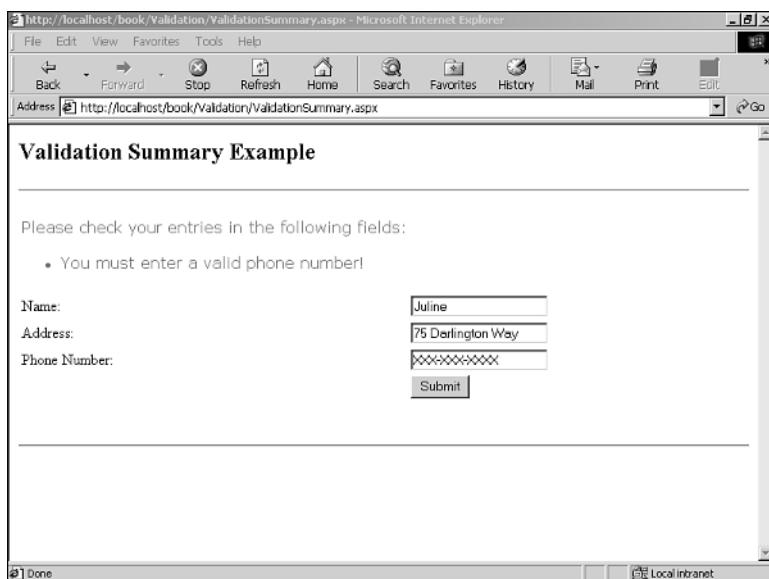


Figure 12.10

The appearance of validationSummary control output with multiple errors.

CHAPTER 13

Data Access with ADO.NET

ADO.NET is the primary data access tool in the Microsoft .NET framework. ADO.NET is designed to be scalable and interoperable, as well as easy to use. For a broad overview of ADO.NET, see Chapter 7, “Understanding Data Access with ADO.NET.” ADO.NET provides three namespaces that enable you to retrieve data from any data source:

- `System.Data`—This namespace provides many of the basic objects on which other classes and namespaces depend, such as `DataSet` and `DataTable`.
- `System.Data.OleDb`—This namespace provides data access to any valid object linking and embedding database (OLE DB) source.
- `System.Data.SqlClient`—This namespace is a managed provider for SQL Server that works for version 7.0 and above. Since it is a managed provider and does not have to operate through an OLE DB layer, it provides a large performance benefit.

The objects in these three namespaces mirror each other. For example, if you know how to use the `SqlClientDataAdapter` object, you will also know how to use the `OleDbDataAdapter` objects because the objects are very similar.

Table 13.1 System.Data.SqlClient Classes at a Glance

Classes	Description
<code>SqlConnection</code> , <code>OleDbConnection</code>	Provides the properties and methods necessary to make a connection to the data source

Table 13.1 *continued*

Classes	Description
SqlCommand, OleDbCommand	Provides the properties and methods necessary to perform operations against the data source (such as retrieving, updating, and deleting data)
SqlParameter, OleDbParameter	Provides the properties and methods necessary to pass parameters to the data source
SqlDataReader, OleDbDataReader	Provides a lightweight yet powerful means of retrieving large amounts of data from the data source
SqlDataAdapter, OleDbDataAdapter	Provides a way to retrieve data from the data source and fill a data set

Using the Connection Object

The first step in retrieving data from a data source is to make a connection to the data source. You do this by using the `Connection` object.

Listings 13.1 and 13.2 show the code you use to open a connection to the Northwind database on a Structured Query Language (SQL) server that uses a managed provider.

Listing 13.1 Using the Connection Object (C#)

```
void Page_Load(Object Source, EventArgs E)
{
    SqlConnection nwConn = new SqlConnection("Data Source=localhost;
        User Id=sa; Password=;Initial Catalog=northwind");

    nwConn.Open();
}
```

Listing 13.2 Using the Connection Object (Visual Basic.NET)

```
Sub Page_Load(Source as Object, E as EventArgs)

    dim nwConn as new SqlConnection("Data Source=localhost; User Id=sa; _
        Password=;Initial Catalog=northwind")

    nwConn.Open()
End Sub
```

The constructor for the `SqlConnection` object accepts a connection string to the data source. After the object is instantiated, the `Open()` method opens the connection.

You use the `Close()` method to close a database connection. The `Close()` method takes no arguments, returns nothing, and is used like this in C#:

```
nwConn.Close();  
and like this in Visual Basic.NET:  
nwConn.Close()
```

**TIP**

You should always open a connection to the database as late as possible in your code and close the connection as quickly as possible. Typically, in an application that connects to a database server, database connections are a relatively scarce resource. By opening a connection as late as possible in the application and closing it as quickly as possible, you will generally improve the performance of an application.

Using the Command Object

When you have an open connection to the data source, you use the `Command` object to perform queries and execute stored procedures. The `Command` object has several different methods for executing different types of queries. Each of the methods is most efficient for the type of query it is meant to execute.

One of the simplest `Command` object methods is the `ExecuteScalar()` method. You use `ExecuteScalar()` to perform a query that returns a single value. Listings 13.3 and 13.4 show how to use the `ExecuteScalar()` method to return the number of items in a table.

Listing 13.3 Using the ExecuteScalar() Method (C#)

```
void Page_Load(Object Source, EventArgs E)  
{  
  
    SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
                                              User Id=sa; Password=;Initial Catalog=northwind");  
  
    nwConn.Open();  
  
    SqlCommand nwCmd = new SqlCommand("SELECT count(*)  
                                      FROM Employees", nwConn);  
  
    nwCmd.ExecuteScalar();  
}
```

Listing 13.4 Using the ExecuteScalar() Method (Visual Basic.NET)

```
Sub Page_Load(Source as Object, E as EventArgs)  
  
    dim nwConn as new SqlConnection("Data Source=localhost; _  
                                    User Id=sa; Password=;Initial Catalog=northwind")  
  
    nwConn.Open()
```

Listing 13.4 continued

```
dim nwCmd as new SqlCommand("SELECT count(*) FROM Employees", nwConn)  
  
    nwCmd.ExecuteScalar()  
End Sub
```

Some queries, such as UPDATE and DELETE queries, do not return any data. A Command object method specifically handles these instances: ExecuteNonQuery(). Listings 13.5 and 13.6 shows how to use this method.

Listing 13.5 Using the ExecuteNonQuery() Method (C#)

```
void Page_Load(Object Source, EventArgs E)  
{  
  
    SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
                                            User Id=sa; Password=;Initial Catalog=northwind");  
  
    nwConn.Open();  
  
    SqlCommand nwCmd = new SqlCommand("DELETE FROM Employees  
                                      WHERE LastName='Tomb'", nwConn);  
  
    nwCmd.ExecuteNonQuery();  
}
```

Listing 13.6 Using the ExecuteNonQuery() Method (Visual Basic.NET)

```
Sub Page_Load(Source as Object, E as EventArgs)  
  
    dim nwConn as new SqlConnection("Data Source=localhost; _  
                                    User Id=sa; Password=;Initial Catalog=northwind")  
  
    nwConn.Open()  
  
    dim nwCmd as new SqlCommand("DELETE FROM Employees _  
                                WHERE LastName='Tomb'", nwConn)  
  
    nwCmd.ExecuteNonQuery()  
End Sub
```

Using the DataReader Object

In previous versions of ASP, you have to step through a recordset and manually build Hypertext Markup Language (HTML) for each record. The .NET framework eliminates this requirement. The .NET framework provides incredible data iteration devices such as list controls that make it very easy to retrieve data from just about any OLE DB data source, plug the data into an object, and automatically format and display each record on the page by using templates.

The DataReader object enables you to step through each item in a dataset and perform an operation. The DataReader object is extremely efficient; it needs only one record in memory at any one time. Thus, the DataReader object is particularly useful for working with large result sets.

Instantiating the DataReader Object

Listings 13.7 and 13.8 show how to use the DataReader object to display records on a Web page.

NOTE

The DataReader object example shown here shows how to use the DataReader object's properties and methods. However, this is technically a better job for a list control than for the DataReader object. Furthermore, if an application requires more functionality than the built-in list controls can provide, you should build your own list control, instead of attempting to use the DataReader object, to manually format HTML. For more information on list controls, see Chapter 11, "Working with ASP.NET List Controls."

Listing 13.7 Using the DataReader Object (C#)

```
<% @Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

<script language="C#" runat="server" >
    void Page_Load(Object Source, EventArgs E)
    {
        string message = "";

        SqlConnection nwConn = new SqlConnection("Data Source=localhost;
                                                User Id=sa; Password=;Initial Catalog=northwind");

        nwConn.Open();

        SqlCommand nwCmd = new SqlCommand("SELECT EmployeeID, LastName,
                                            FirstName FROM Employees", nwConn);

        SqlDataReader nwReader = nwCmd.ExecuteReader();

        int EmployeeIDColumn = nwReader.GetOrdinal("EmployeeID");
        int LastNameColumn = nwReader.GetOrdinal("LastName");
        int FirstNameColumn = nwReader.GetOrdinal("FirstName");
```

Listing 13.7 continued

```
while (nwReader.Read()) {  
    message = message + nwReader.GetInt32(EmployeeIDColumn).ToString()  
        + " " + nwReader.GetString(LastNameColumn)  
        + ", " + nwReader.GetString(FirstNameColumn)  
        + "<BR>";  
}  
  
nwReader.Close();  
  
msg.Text = message;  
  
}  
</script>  
  
</HEAD>  
<BODY>  
  
<h1>The DataReader Object</h1>  
<hr>  
  
<form runat="server" id=form1 name=form1>  
    <asp:Label id=msg runat="server"></asp:Label>  
</form>  
<hr>  
  
</BODY>  
</HTML>
```

Listing 13.8 Using the DataReader Object (Visual Basic.NET)

```
<% @Page Language="VB" %>  
<%@ Import Namespace="System.Data" %>  
<%@ Import Namespace="System.Data.SqlClient" %>  
  
<HTML>  
<HEAD>  
    <LINK rel="stylesheet" type="text/css" href="Main.css">  
    <!-- End Style Sheet -->  
  
<script language="VB" runat="server" >  
    Sub Page_Load(Source as Object, E as EventArgs)  
  
        dim message as string  
        dim EmployeeIDColumn as Int32  
        dim LastNameColumn as Int32  
        dim FirstNameColumn as Int32  
  
        dim nwConn as new SqlConnection("Data Source=localhost; _  
                                         User Id=sa; Password=;Initial Catalog=northwind")
```

Listing 13.8 continued

```

nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID, LastName,
                           FirstName FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

EmployeeIDColumn = nwReader.GetOrdinal("EmployeeID")
LastNameColumn = nwReader.GetOrdinal("LastName")
FirstNameColumn = nwReader.GetOrdinal("FirstName")

While (nwReader.Read())
    message = message _
        + nwReader.GetInt32(EmployeeIDColumn).ToString() _
        + " " + nwReader.GetString(LastNameColumn) + ", " _
        + nwReader.GetString(FirstNameColumn) + "<BR>"
End While

nwReader.Close()

msg.Text = message

End Sub
</script>

</HEAD>
<BODY>

<h1>The DataReader Object</h1>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:Label id=msg runat="server"></asp:Label>
</form>
<hr>

</BODY>
</HTML>

```

NOTE

Because in this example, the user is connecting to a SQL server that uses the SQL managed provider, the example uses the `SqlDataReader` object. Because `System.Data.OleDb` is nearly identical to `System.Data.SqlClient`, an example using a `DataReader` object with OLE DB would be very similar—it would require only that the object names change.

Notice in Listing 13.7 and 13.8 that to use a `DataReader` object, you must first create `Connection` and `Command` objects. The `ExecuteReader()` method of the `Command` object returns an object of type `SqlDataReader`. With the `DataReader` object connected to the data source and retrieving data, you can now read the results.

Stepping Through a Result Set with `Read()`

Now that you have an instance of a `DataReader` object, you can step through the result set returned from the SQL query in the `SqlCommand` object. You do this by using the `Read()` method of the `DataReader` object. If there is more records in the result set, the `Read()` method will advance to the next record and return `True`. If there are no additional records, the `Read()` method returns `False`.

The `DataReader` object also contains several methods for retrieving the data from the current dataset. In fact, there is a unique method for each possible type of data that SQL server can return. However, the SQL data types and the Microsoft .NET framework data types do not align perfectly. For instance, `char`, `varchar` and `nvarchar`s are all considered strings from the framework's point of view.

The `DataReader` object's data retrieval methods all reference the data by ordinal number. To ensure that you are retrieving the correct data, you can use the `GetOrdinal()` method. When you pass `GetOrdinal()` a field name that is used in a query, it returns the field's ordinal position in the `DataReader` object.

Also, because the `DataReader` object stores these values as a collection, the data access portion of the code from Listings 13.7 and 13.8 could look like the code in Listings 13.9 and 13.10. Using the method of data access from Listings 13.7 and 13.8 is theoretically slower than using the specific data access method for the specific data type being returned. However, the readability of the code is enhanced significantly.

Listing 13.9 An Alternative Way to Access a DataReader Object's Data (C#)

```
void Page_Load(Object Source, EventArgs E)
{
    string message = " ";

    SqlConnection nwConn = new SqlConnection("Data Source=localhost,
                                              User Id=sa; Password=;Initial Catalog=northwind");

    nwConn.Open();

    SqlCommand nwCmd = new SqlCommand("SELECT EmployeeID, LastName, FirstName
                                     FROM Employees", nwConn);

    SqlDataReader nwReader = nwCmd.ExecuteReader();

    while (nwReader.Read()) {
        message = message + nwReader["EmployeeID"] + " "
```

Listing 13.9 continued

```

        + nwReader[ "LastName" ] + " , "
        + nwReader[ "FirstName" ] + "<BR>";
    }

nwReader.Close();

msg.Text = message;

}

```

Listing 13.10 An Alternative Way to Access a DataReader Object's Data (Visual Basic.NET)

```

Sub Page_Load(Source as Object, E as EventArgs)
{
    dim message as string

    dim nwConn as new SqlConnection("Data Source=localhost; User Id=sa; _
                                    Password=;Initial Catalog=northwind")

    nwConn.Open()

    dim nwCmd as new SqlCommand("SELECT EmployeeID, LastName, FirstName _ 
                                FROM Employees", nwConn)

    dim nwReader as object
    SqlDataReader nwReader = nwCmd.ExecuteReader()

    While (nwReader.Read())
        message = message + nwReader[ "EmployeeID" ] + " " _
                    + nwReader[ "LastName" ] + " , " _
                    + nwReader[ "FirstName" ] + "<BR>"
    End While

    nwReader.Close()

    msg.Text = message
}

End Sub

```

You use the `DataAdapter` object to fill a data table with the results of a query.

Listings 13.11 and 13.12 show a very simple example of using a `DataAdapter` object to fill a dataset. First, a connection is made to the SQL database. Then, a `SqlDataAdapter` object is created with simple select query, using the `nwConn` connection. The `dsEmployees` dataset is then created and filled with the results of the query. The dataset is then bound to a `DataGridView` object for display on the page.

Listing 13.11 Using the DataAdapter Object (C#)

```
<% @Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {

            SqlConnection nwConn = new SqlConnection("Data Source=localhost;
                User Id=sa; Password=;Initial Catalog=northwind");
            nwConn.Open();

            SqlDataAdapter nwAdapter = new SqlDataAdapter("SELECT EmployeeID,
                LastName, FirstName FROM Employees", nwConn);

            DataSet dsEmployees = new DataSet();
            nwAdapter.Fill(dsEmployees, "Employees");

            employees.DataSource = dsEmployees.Tables["Employees"];
            employees.DataBind();
        }
    </script>

</HEAD>
<BODY>

<h1>Using The DataAdapter Object</h1>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:DataGrid id="employees" runat="server"></asp:DataGrid>
</form>
<hr>

</BODY>
</HTML>
```

Listing 13.12 Using the DataAdapter Object (Visual Basic.NET)

```
<% @Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
```

Listing 13.12 continued

```

<HTML>
<HEAD>
<LINK rel="stylesheet" type="text/css" href="Main.css">
<!-- End Style Sheet -->

<script language="VB" runat="server" >
Sub Page_Load(Source as Object, E as EventArgs)

    dim nwConn as new SqlConnection("Data Source=localhost; _
                                    User Id=sa; Password=;Initial Catalog=northwind")
    nwConn.Open()
    dim nwAdapter as object
    nwAdapter = new SqlDataAdapter("SELECT EmployeeID, LastName, _
                                    FirstName FROM Employees", nwConn)

    dim dsEmployees as new DataSet()
    nwAdapter.Fill(dsEmployees, "Employees")

    employees.DataSource = dsEmployees.Tables("Employees")
    employees.DataBind()
End Sub
</script>

</HEAD>
<BODY>

<h1>Using The DataAdapter Object</h1>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:DataGrid id="employees" runat="server"></asp:DataGrid>
</form>
<hr>

</BODY>
</HTML>

```

In Listings 13.11 and 13.12, the `DataAdapter` object automatically builds a `Command` object in order to make the query. However, it might be more convenient to create a `Command` object manually and then programmatically assign it to the `DataAdapter` object, as shown in Listings 13.13 and 13.14.

Listing 13.13 Assigning a Command Object to a DataAdapter Object (C#)

```

<% @Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>

```

Listing 13.13 continued

```
<LINK rel="stylesheet" type="text/css" href="Main.css">
<!-- End Style Sheet -->

<script language="C#" runat="server" >
    void Page_Load(Object Source, EventArgs E)
    {

        SqlConnection nwConn = new SqlConnection("Data Source=localhost;
                                                User Id=sa; Password=;Initial Catalog=northwind");
        nwConn.Open();

        SqlCommand nwCommand = new SqlCommand("SELECT EmployeeID, LastName,
                                                FirstName FROM Employees", nwConn);

        SqlDataAdapter nwAdapter = new SqlDataAdapter();

        nwAdapter.SelectCommand = nwCommand;

        DataSet dsEmployees = new DataSet();
        nwAdapter.Fill(dsEmployees, "Employees");

        employees.DataSource = dsEmployees.Tables["Employees"];
        employees.DataBind();
    }
</script>

</HEAD>
<BODY>

<h1>Using The DataAdapter Object</h1>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:DataGrid id="employees" runat="server"></asp:DataGrid>
</form>
<hr>

</BODY>
</HTML>
```

Listing 13.14 Assigning a Command Object to a DataAdapter Object (Visual Basic.NET)

```
<% @Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
```

Listing 13.14 continued

```

<!-- End Style Sheet -->

<script language="VB" runat="server" >
Sub Page_Load(Source as Object, E as EventArgs)

    dim nwConn as new SqlConnection("Data Source=localhost; _
                                    User Id=sa; Password=;Initial Catalog=northwind")
    nwConn.Open()

    dim nwCommand as new SqlCommand("SELECT EmployeeID, LastName, _
                                    FirstName FROM Employees", nwConn)

    dim nwAdapter as new SqlDataAdapter()

    nwAdapter.SelectCommand = nwCommand

    dim dsEmployees as new DataSet()
    nwAdapter.Fill(dsEmployees, "Employees")

    employees.DataSource = dsEmployees.Tables("Employees")
    employees.DataBind()
End Sub
</script>

</HEAD>
<BODY>

<h1>Using The DataAdapter Object</h1>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:DataGrid id="employees" runat="server"></asp:DataGrid>
</form>
<hr>

</BODY>
</HTML>

```

Working with Stored Procedures

Working with stored procedures is a common task in building Microsoft .NET applications, as it is in Windows DNA programming, using ADO. In fact, anyone who is familiar with working with stored procedures in ADO will feel very comfortable working with stored procedures in ADO.NET.

Retrieving a Dataset

Retrieving a dataset from a stored procedure is fairly straightforward and can be done in several ways. Listings 13.15 and 13.16 show one way to retrieve a dataset from a stored procedure and bind it to a DataGrid object.

Listing 13.15 An Example of Using a Simple Stored Procedure (C#)

```
<% @Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {

            SqlConnection nwConn = new SqlConnection("Data Source=localhost;
                User Id=sa; Password=;Initial Catalog=northwind");

            SqlDataAdapter nwAdapter = new SqlDataAdapter("exec [Ten Most
                Expensive Products]", nwConn);

            DataSet report = new DataSet();
            nwAdapter.Fill(report, "MostExpensiveItems");

            salesReport.DataSource = report.Tables["MostExpensiveItems"];
            salesReport.DataBind();

        }
    </script>

</HEAD>
<BODY>

<h1>Returning a DataSet From a Stored Procedure</h1>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:DataGrid id=salesReport runat=server></asp:DataGrid>
</form>
<hr>

</BODY>
</HTML>
```

Listing 13.16 An Example of Using a Simple Stored Procedure (Visual Basic.NET)

```
<% @Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
```

Listing 13.16 continued

```

<HTML>
<HEAD>
<LINK rel="stylesheet" type="text/css" href="Main.css">
<!-- End Style Sheet -->

<script language="VB" runat="server" >
Sub Page_Load(Source as Object, E as EventArgs)

    dim nwConn as new SqlConnection("Data Source=localhost; _
                                    User Id=sa; Password=;Initial Catalog=northwind")

    dim nwAdapter as new SqlDataAdapter("exec [Ten Most _
                                         Expensive Products]", nwConn)

    dim report as new DataSet()
    nwAdapter.Fill(report, "MostExpensiveItems")

    salesReport.DataSource = report.Tables("MostExpensiveItems")
    salesReport.DataBind()

End Sub
</script>

</HEAD>
<BODY>

<h1>Returning a DataSet From a Stored Procedure</h1>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:DataGrid id=salesReport runat=server></asp:DataGrid>
</form>
<hr>

</BODY>
</HTML>

```

The example in Listings 13.15 and 13.16 shows one way of retrieving a dataset from a stored procedure. However, this is not the best way. If the stored procedure required input or output parameters, it would be difficult to build and maintain the query string, and the readability of the code would suffer.

The example in Listings 13.17 and 13.18 shows a better way to return a simple dataset; this example returns a dataset from a stored procedure that has no parameters. A `SqlCommand` object is created, and the `CommandType` property is set to `StoredProcedure`. This `SqlCommand` object is then assigned to the `SelectCommand` property of the `DataAdapter` object. In this example, the managed provider knows that it is working with a stored procedure. In Listings 13.15 and 13.16, the call to the stored procedure is treated the same way as any other query.

Listing 13.17 Retrieving a Dataset from a Stored Procedure (C#)

```
<% @Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

<script language="C#" runat="server" >
    void Page_Load(Object Source, EventArgs E)
    {
        //create and open connection to the database
        SqlConnection nwConn = new SqlConnection("Data Source=localhost;
                                                User Id=sa; Password=;Initial Catalog=northwind");
        nwConn.Open();

        //create command object and set type of command to stored procedure
        SqlCommand nwCommand = new SqlCommand("Ten Most Expensive Products",
                                              nwConn);
        nwCommand.CommandType = CommandType.StoredProcedure;

        //create adapter object and assign command object above to adapter
        SqlDataAdapter nwAdapter = new SqlDataAdapter();
        nwAdapter.SelectCommand = nwCommand;

        //create and fill dataset
        DataSet report = new DataSet();
        nwAdapter.Fill(report, "MostExpensiveItems");

        //Bind to datagrid
        salesReport.DataSource = report.Tables["MostExpensiveItems"];
        salesReport.DataBind();
    }
</script>

</HEAD>
<BODY>

<h1>Returning a DataSet From a Stored Procedure</h1>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:DataGrid id=salesReport runat=server></asp:DataGrid>
</form>
<hr>
```

Listing 13.17 continued

```
</BODY>
</HTML>
```

Listing 13.18 Retrieving a Dataset from a Stored Procedure (Visual Basic.NET)

```
<% @Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >
        Sub Page_Load(Source as Object, E as EventArgs)

            'create and open connection to the database
            dim nwConn as new SqlConnection("Data Source=localhost; _
                User Id=sa; Password=;Initial Catalog=northwind")
            nwConn.Open()

            'create command object and set type of command to stored procedure
            dim nwCommand as new SqlCommand("Ten Most Expensive Products", _
                nwConn)
            nwCommand.CommandType = CommandType.StoredProcedure

            'create adapter object and assign command object above to adapter
            dim nwAdapter as object
            nwAdapter = new SqlDataAdapter()
            nwAdapter.SelectCommand = nwCommand

            'create and fill dataset
            dim report as new DataSet()
            nwAdapter.Fill(report, "MostExpensiveItems")

            'Bind to datagrid
            salesReport.DataSource = report.Tables("MostExpensiveItems")
            salesReport.DataBind()
        End Sub
    </script>

</HEAD>
<BODY>

<h1>Returning a DataSet From a Stored Procedure</h1>
<hr>
```

Listing 13.18 continued

```
<form runat="server" id=form1 name=form1>
  <asp:DataGrid id=salesReport runat=server></asp:DataGrid>
</form>
<hr>

</BODY>
</HTML>
```

Using Parameters

With many stored procedures, parameters are used to pass arguments from the application into the stored procedure. For instance, parameters enable you to write a single stored procedure that returns category information for a specific CategoryID procedure (passed in as a parameter) rather than having to write a separate stored procedure for each category in the database.

Input Parameters

The `SqlParameter` object in the `System.Data.SqlClient` namespace contains the properties and methods that are necessary to work with stored procedure parameters.

Listings 13.19 and 13.20 show a simple example that calls a stored procedure with a single parameter. As usual, the `Connection` and `Command` objects are created and initialized, and the `CommandType` property is set to `StoredProcedure`. Then, before the `nwAdapter.Fill()` method is called, the stored procedure parameter is set up.

First, a variable named `sprocParams`, of type `SqlParameter`, is created. Then, a new `SqlParameter` variable is added to the parameter collection of the `nwCommand` object. On the same line, the parameter collection is assigned to the `SqlParameter` variable. Next, the direction of the parameter is set to input (from the point of view of the stored procedure itself). Finally, the value of the parameter is set.

Of course, this example shows only one way to handle parameters. The `sprocParams` object is created mainly for readability and ease of use. Having to reference `nwCommand.Parameters.Direction` is much more difficult than simply referencing `sprocParams.Direction`.

Listing 13.19 Using Input Parameters (C#)

```
<% @Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
  <LINK rel="stylesheet" type="text/css" href="Main.css">
  <!-- End Style Sheet -->

<script language="C#" runat="server" >
  void Page_Load(Object Source, EventArgs E)
```

Listing 13.19 continued

```
{  
    //Create Connection  
    SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
                                              User Id=sa; Password=;Initial Catalog=northwind");  
  
    //Create Command  
    SqlCommand nwCommand = new SqlCommand("SalesByCategory", nwConn);  
  
    //Set as Stored Procedure  
    nwCommand.CommandType = CommandType.StoredProcedure;  
  
    //Set up parameters  
    SqlParameter sprocParams = null;  
    sprocParams = nwCommand.Parameters.Add(  
        new SqlParameter("@CategoryName", SqlDbType.NVarChar, 15));  
    sprocParams.Direction = ParameterDirection.Input;  
    sprocParams.Value = "Beverages";  
  
    //Create Adapter and set to current command object  
    SqlDataAdapter nwAdapter = new SqlDataAdapter();  
    nwAdapter.SelectCommand = nwCommand;  
  
    //Fill Dataset  
    DataSet report = new DataSet();  
    nwAdapter.Fill(report, "SalesByCategory");  
  
    //Bind to DataGrid  
    salesReport.DataSource = report.Tables["SalesByCategory"];  
    salesReport.DataBind();  
  
}  
</script>  
  
</HEAD>  
<BODY>  
  
<h1>Returning Values From a Stored Procedure</h1>  
<hr>  
  
<form runat="server" id=form1 name=form1>  
    <asp:DataGrid id=salesReport runat="server"></asp:DataGrid>  
</form>  
<hr>  
  
</BODY>  
</HTML>
```

Listing 13.20 Using Input Parameters (Visual Basic.NET)

```
<% @Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

<script language="VB" runat="server" >
Sub Page_Load(Source as Object, E as EventArgs)

    'Create Connection
    dim nwConn as new SqlConnection("Data Source=localhost; User Id=sa; _
                                    Password=;Initial Catalog=northwind")
    nwConn.Open()

    'Create Command
    dim nwCommand as new SqlCommand("SalesByCategory", nwConn)

    'Set as Stored Procedure
    nwCommand.CommandType = CommandType.StoredProcedure

    'Set up parameters
    dim sprocParams as new SqlParameter()
    sprocParams = nwCommand.Parameters.Add( _
                new SqlParameter("@CategoryName", SqlDbType.NVarChar, 15))
    sprocParams.Direction = ParameterDirection.Input
    sprocParams.Value = "Beverages"

    'Create Adapter and set to current command object
    dim nwAdapter as new SqlDataAdapter()
    nwAdapter.SelectCommand = nwCommand

    'Fill Dataset
    dim report as new DataSet()
    nwAdapter.Fill(report, "SalesByCategory")

    'Bind to DataGrid
    salesReport.DataSource = report.Tables("SalesByCategory")
    salesReport.DataBind()
End Sub
</script>

</HEAD>
<BODY>
```

Listing 13.20 continued

```
<h1>Returning Values From a Stored Procedure</h1>
<hr>

<form runat="server" id=form1 name=form1>
  <asp:DataGrid id=salesReport runat="server"></asp:DataGrid>
</form>
<hr>

</BODY>
</HTML>
```

Output Parameters

For the most part, you handle output parameters in the same was as input parameters. They are built in the same manner as input parameters, except that `sprocParam.Direction` is set to `ParameterDirection.Output`. The `Value` property of the output parameter is available after the `Command` object that contains the call to the stored procedure is run.

A common task in writing a Windows DNA application is to add a new record to a table and return the value of the `Identity` field for the newly added record. For instance, Listing 13.21 shows a new stored procedure named `AddNewEmployee` that is added to the Northwind database in order to easily add a new employee to the database. You simply run this script from the Query Analyzer tool in order to create the stored procedure in the Northwind database. Note that the stored procedure accepts six arguments and returns the identity of the newly added record. In this case, the value is an integer.

Listing 13.21 Generating the AddNewEmployee Stored Procedure

```
CREATE PROCEDURE [AddNewEmployee]
(
    @LastName nvarchar(20),
    @FirstName nvarchar(10),
    @HireDate datetime,
    @Address nvarchar(60),
    @City nvarchar(20),
    @Extension nvarchar(4),
    @retval int OUTPUT
)
AS

INSERT INTO Employees
(
    LastName,
    FirstName,
    HireDate,
    Address,
    City,
    Extension
```

Listing 13.21 continued

```
)  
VALUES  
(  
    @LastName,  
    @FirstName,  
    @HireDate,  
    @Address,  
    @City,  
    @Extension  
)  
  
select @retval = @@identity  
GO
```

Listings 13.22 and 13.23 show a Web form that adds a new employee to the Northwind database. The form uses the stored procedure in Listing 13.21 to perform this operation.

Listing 13.22 Using Input and Output Parameters (C#)

```
<% @Page Language="C#" Debug="True" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

<script language="C#" runat="server" >
    void Page_Load(Object Source, EventArgs E)
    {
        if(IsPostBack==true) {
            //Create and Open Connection
            SqlConnection nwConn = new SqlConnection(
                "Data Source=localhost;UID=sa;PWD=;Initial Catalog=northwind;");
            nwConn.Open();

            //Create Command
            SqlCommand nwCmd = new SqlCommand("AddNewEmployee", nwConn);
            nwCmd.CommandType = CommandType.StoredProcedure;

            //Setup the following Input and Output Parameters
            /*
                Name          Size
                - - - - -     - - - - -
                @LastName      20
                @FirstName     10
            */
        }
    }
</script>

```

Listing 13.22 continued

```

        @HireDate      8
        @Address       60
        @City          15
        @Extension     4
    */

SqlParameter nwParam = null;

nwParam = nwCmd.Parameters.Add(new SqlParameter("@LastName",
                                              SqlDbType.NVarChar, 20));
nwParam.Direction = ParameterDirection.Input;
nwParam.Value = sLastName.Text;

nwParam = nwCmd.Parameters.Add(new SqlParameter("@FirstName",
                                              SqlDbType.NVarChar, 10));
nwParam.Direction = ParameterDirection.Input;
nwParam.Value = sFirstName.Text;

nwParam = nwCmd.Parameters.Add(new SqlParameter("@HireDate",
                                              SqlDbType.NVarChar, 8));
nwParam.Direction = ParameterDirection.Input;
nwParam.Value = sHireDate.Text;

nwParam = nwCmd.Parameters.Add(new SqlParameter("@Address",
                                              SqlDbType.NVarChar, 60));
nwParam.Direction = ParameterDirection.Input;
nwParam.Value = sAddress.Text;

nwParam = nwCmd.Parameters.Add(new SqlParameter("@City",
                                              SqlDbType.NVarChar, 15));
nwParam.Direction = ParameterDirection.Input;
nwParam.Value = sCity.Text;

nwParam = nwCmd.Parameters.Add(new SqlParameter("@Extension",
                                              SqlDbType.NVarChar, 4));
nwParam.Direction = ParameterDirection.Input;
nwParam.Value = sExtension.Text;

nwParam = nwCmd.Parameters.Add(new SqlParameter("@retval",
                                              SqlDbType.Int, 4));
nwParam.Direction = ParameterDirection.Output;

nwCmd.ExecuteNonQuery();
msg.Text = "New Employee Added with EmployeeID '"
            + nwParam.Value + "'";
}

</script>

</HEAD>

```

Listing 13.22 continued

```
<BODY>

<h1>Using Stored Procedures With Parameters (with output)</h1>
<hr>

<form runat="server" id=form1 name=form1>

    <asp:Label id=msg runat="server">
        Please Enter The Following Information
        To Add A New Employee To The NorthWinds DB
    </asp:Label>

    <p>
    <table>
        <tr>
            <td>
                First Name:
            </td>
            <td>
                <asp:TextBox id=sFirstName runat=server></asp:TextBox>
            </td>
        </tr>
        <tr>
            <td>
                Last Name:
            </td>
            <td>
                <asp:TextBox id=sLastName runat=server></asp:TextBox>
            </td>
        </tr>
        <tr>
            <td>
                Hire Date:
            </td>
            <td>
                <asp:TextBox id=sHireDate runat=server></asp:TextBox>
            </td>
        </tr>
        <tr>
            <td>
                Address:
            </td>
            <td>
                <asp:TextBox id=sAddress runat=server></asp:TextBox>
            </td>
        </tr>
        <tr>
            <td>
                City:
            </td>
        </tr>
```

Listing 13.22 continued

```

</td>
<td>
    <asp:TextBox id=sCity runat=server></asp:TextBox>
</td>
</tr>
<tr>
    <td>
        Extension:
    </td>
    <td>
        <asp:TextBox id=sExtension runat=server></asp:TextBox>
    </td>
</tr>
<tr>
    <td colspan=2>
        <input type=submit id=submit1 name=submit1>
    </td>
</tr>
</table>
</p>
</form>
<hr>

</BODY>
</HTML>

```

Listing 13.23 Using Input and Output Parameters (Visual Basic.NET)

```

<% @Page Language="VB" Debug="True" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

<script language="VB" runat="server" >
Sub Page_Load(Source as Object, E as EventArgs)
    if IsPostBack = true
        'Create and Open Connection
        dim nwConn as new SqlConnection( _
            "Data Source=localhost;UID=sa;PWD=;Initial Catalog=northwind;" )
        nwConn.Open()

        'Create Command
        dim nwCmd as new SqlCommand("AddNewEmployee", nwConn)
        nwCmd.CommandType = CommandType.StoredProcedure

```

Listing 13.23 continued

```
'Setup the following Input and Output Parameters

'Name          Size
'-- -- -- -- --
'@LastName      20
'@FirstName     10
'@HireDate      8
'@Address       60
'@City          15
'@Extension     4

dim nwParam as new SqlParameter

nwParam = nwCmd.Parameters.Add(new SqlParameter("@LastName", _
                                         SqlDbType.NVarChar, 20))
nwParam.Direction = ParameterDirection.Input
nwParam.Value = sLastName.Text

nwParam = nwCmd.Parameters.Add(new SqlParameter("@FirstName", _
                                         SqlDbType.NVarChar, 10))
nwParam.Direction = ParameterDirection.Input
nwParam.Value = sFirstName.Text

nwParam = nwCmd.Parameters.Add(new SqlParameter("@HireDate", _
                                         SqlDbType.NVarChar, 8))
nwParam.Direction = ParameterDirection.Input
nwParam.Value = sHireDate.Text

nwParam = nwCmd.Parameters.Add(new SqlParameter("@Address", _
                                         SqlDbType.NVarChar, 60))
nwParam.Direction = ParameterDirection.Input
nwParam.Value = sAddress.Text

nwParam = nwCmd.Parameters.Add(new SqlParameter("@City", _
                                         SqlDbType.NVarChar, 15))
nwParam.Direction = ParameterDirection.Input
nwParam.Value = sCity.Text

nwParam = nwCmd.Parameters.Add(new SqlParameter("@Extension", _
                                         SqlDbType.NVarChar, 4))
nwParam.Direction = ParameterDirection.Input
nwParam.Value = sExtension.Text

nwParam = nwCmd.Parameters.Add(new SqlParameter("@retval", _
                                         SqlDbType.Int, 4))
nwParam.Direction = ParameterDirection.Output

nwCmd.ExecuteNonQuery()
msg.Text = "New Employee Added with EmployeeID '" + nwParam.Value + "'"
```

Listing 13.23 continued

```
end if
End Sub
</script>

</HEAD>
<BODY>

<h1>Using Stored Procedures With Parameters (with output)</h1>
<hr>

<form runat="server" id=form1 name=form1>

    <asp:Label id=msg runat="server">
        Please Enter The Following Information
        To Add A New Employee To The NorthWinds DB
    </asp:Label>

    <p>
    <table>
        <tr>
            <td>
                First Name:
            </td>
            <td>
                <asp:TextBox id=sFirstName runat=server></asp:TextBox>
            </td>
        </tr>
        <tr>
            <td>
                Last Name:
            </td>
            <td>
                <asp:TextBox id=sLastName runat=server></asp:TextBox>
            </td>
        </tr>
        <tr>
            <td>
                Hire Date:
            </td>
            <td>
                <asp:TextBox id=sHireDate runat=server></asp:TextBox>
            </td>
        </tr>
        <tr>
            <td>
                Address:
            </td>
            <td>
                <asp:TextBox id=sAddress runat=server></asp:TextBox>
            </td>
        </tr>
    </table>
</form>
```

Listing 13.23 continued

```
</td>
</tr>
<tr>
<td>
    City:
</td>
<td>
    <asp:TextBox id=sCity runat=server></asp:TextBox>
</td>
</tr>
<tr>
<td>
    Extension:
</td>
<td>
    <asp:TextBox id=sExtension runat=server></asp:TextBox>
</td>
</tr>
<tr>
<td colspan=2>
    <input type=submit id=submit1 name=submit1>
</td>
</tr>
</table>
</p>
</form>
<hr>

</BODY>
</HTML>
```

Working with Transactions

Transactions enable you to treat a set of SQL queries as one logical operation. Transactions maintain the consistency and integrity of data by ensuring that either all queries fail or all queries succeed.

Beginning a Transaction

To begin a transaction, you first need to instantiate a connection object. The `BeginTransaction()` method of the connection object places that connection under a transaction and returns a `SqlTransaction` object that can be used to control the transaction throughout its life cycle.

After the transaction has been started, all operations performed on the database through that `Connection` object will be part of the same transaction. The code in Listings 13.24 and 13.25 instantiates a connection and begins a transaction.

Listing 13.24 Beginning a Transaction (C#)

```
//Create and Open Connection  
SqlConnection nwConn = new SqlConnection(  
    "Data Source=localhost;Initial Catalog=northwind;UID=sa;PWD=");  
nwConn.Open();  
  
//Begin Transaction  
SqlTransaction nwTrans = nwConn.BeginTransaction("RemoveEmployees");
```

Listing 13.25 Beginning a Transaction (Visual Basic.NET)

```
'Create and Open Connection
dim nwConn as new SqlConnection(
    "Data Source=localhost;Initial Catalog=northwind;UID=sa;PWD=;")
nwConn.Open()

'Begin Transaction
dim nwTrans as object
nwTrans = nwConn.BeginTransaction("RemoveEmployees")
```

Rolling Back a Transaction

Rolling back a transaction means undoing all changes performed on the data since the beginning of the transaction. Rolling back a transaction is very simple. The RollBack() method of the transaction object disregards all changes to the database since the beginning of the transaction. Listings 13.26 and 13.27 demonstrate how this method is used.

Listing 13.26 Rolling Back a Transaction (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
                                         User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
//Begin Transaction  
SqlTransaction nwTrans = nwConn.BeginTransaction(IsolationLevel.Serializable,  
                                                 "RemoveEmployees");  
  
//Delete Employee Territory  
SqlCommand nwCmd = new SqlCommand("", nwConn, nwTrans);  
nwCmd.CommandText = "DELETE FROM EmployeeTerritories WHERE EmployeeID=50";  
nwCmd.ExecuteNonQuery();  
  
//Rollback Transaction to Save Point  
nwTrans.Rollback();
```

Listing 13.27 Rolling Back a Transaction (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; User Id=sa; _  
    Password=Initial Catalog=northwind")
```

Listing 13.27 continued

```
nwConn.Open()

'Begin Transaction
dim nwTrans as object
nwTrans = nwConn.BeginTransaction(IsolationLevel.Serializable, _
                    "RemoveEmployees")

'Delete Employee Territory
dim nwCmd as new SqlCommand("", nwConn, nwTrans)
nwCmd.CommandText = "DELETE FROM EmployeeTerritories WHERE EmployeeID=50"
nwCmd.ExecuteNonQuery()

'Rollback Transaction to Save Point
nwTrans.Rollback()
```

Rolling Back a Transaction to a Saved Point

When working with transactions, it is sometimes desirable to bookmark, or save, a transaction at a certain point. It is then possible to roll a transaction back to any saved point, and not have to cancel the entire transaction. Listings 13.28 and 13.29 show how to roll a transaction back to a saved point.

Listing 13.28 Saving a Transaction (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
                                         User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

//Begin Transaction
SqlTransaction nwTrans = nwConn.BeginTransaction(IsolationLevel.Serializable,
                                                "RemoveEmployees");

//Delete Employee Territory
SqlCommand nwCmd = new SqlCommand("", nwConn, nwTrans);
nwCmd.CommandText = "DELETE FROM EmployeeTerritories WHERE EmployeeID=50";
nwCmd.ExecuteNonQuery();

//Save Transaction
nwTrans.Save("TerritoriesRemoved");

//Delete An Employee
nwCmd.CommandText = "DELETE FROM Employees WHERE EmployeeID=60";
nwCmd.ExecuteNonQuery();

//Rollback Transaction to Save Point
nwTrans.Rollback("TerritoriesRemoved");

nwTrans.Commit();
```

Listing 13.29 Saving a Transaction (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; User Id=sa; _  
                                Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
'Begin Transaction  
dim nwTrans as object  
nwTrans = nwConn.BeginTransaction(IsolationLevel.Serializable, _  
                                    "RemoveEmployees")  
  
'Delete Employee Territory  
dim nwCmd as new SqlCommand("", nwConn, nwTrans)  
nwCmd.CommandText = "DELETE FROM EmployeeTerritories WHERE EmployeeID=50"  
nwCmd.ExecuteNonQuery()  
  
'Save Transaction  
nwTrans.Save("TerritoriesRemoved")  
  
'Delete An Employee  
nwCmd.CommandText = "DELETE FROM Employees WHERE EmployeeID=60"  
nwCmd.ExecuteNonQuery()  
  
'Rollback Transaction to Save Point  
nwTrans.Rollback("TerritoriesRemoved")  
  
nwTrans.Commit()
```

Committing a Transaction

When all success conditions of a transaction are met and you are ready to commit the changes to the database, you use the `Commit()` method of the transaction object. The transaction will still be committed if you do not explicitly use the `Commit()` method, but you will see a large performance hit. Therefore, explicitly calling the `Commit()` method is a good practice. Listings 13.28 and 13.29 shows how this property is used.

CHAPTER 14

Building Components for ASP.NET

Components are an important part in developing applications in the Microsoft .NET framework. In this chapter we will discuss what components are, the benefits they offer, and how components are built in the Microsoft .NET framework. At the end of the chapter, we will discuss how to create applications that interoperate with existing Component Object Model (COM) components as well as the services offered by COM+.

What Are Components?

A component, referred to as an *assembly* in Microsoft .NET, is a self-contained library that contains a collection of types and resources. A component typically represents an object, and it includes a number of properties and methods that can be used to manipulate the object. In Microsoft .NET, assemblies represent a vital backbone, forming the fundamental unit of development.

NOTE

Components created with the Microsoft .NET Framework are known as *assemblies*. We stuck with the term *components* throughout this chapter to make the chapter easier to understand.

Benefits of Using Components

Components provide benefits in all phases of the development life cycle, including the design, implementation, and maintenance phases. This section outlines many of these benefits.

Design Benefits

Abstraction is a very important part of object-oriented programming. It is the concept of exposing only the properties and methods that are necessary, hiding the implementation details and the underlying data. This prevents client applications from changing the data within the class without the class knowing, which could potentially introduce errors into the application.

Delegation is another benefit that components provide. By dividing the application into a number of self-contained units, teams can easily develop in parallel on different portions of an application. As team members are designing an application, they can decide which methods and properties the component should expose. Team members that will be referencing that component can then write the necessary calls, knowing that their code will work once the component is complete.

Implementation Benefits

By helping teams work in parallel without stepping on each other's work, components can save time and money during the implementation phase. This is done since using components forces a structured development approach. Taking the time up front to organize the application to use components provides a good roadmap during the implementation phase. In the same way, components can also save time and money by reducing the number of bugs that are introduced into the application, thus preventing the time that must be spent later to find and fix problems.

One of the most obvious benefits of using components is that they provide a very good way to reuse code. You can share common tasks between applications by using a component.

Maintenance Benefits

The benefits that components bring the maintenance phase relate to stability, upgradability, and extensibility. You gain stability because components are well-organized self-contained units, and therefore any changes and modifications to other parts of an application will be less likely to affect the component. Upgradability is similar to stability except that it refers to the ability to upgrade the component without breaking the rest of the application. Finally, extensibility refers to the ability to make additions to the component without breaking the rest of the application.

Microsoft Windows DNA

A concept that came about in the days of COM and COM+ was Windows Distributed Internet Applications (DNA). Windows DNA provided a roadmap that showed developers the recommended way to build enterprise applications with the COM, COM+, and ASP technologies.

Windows DNA involved the concept of dividing an application into three tiers: the database access layer (DAL), the business logic layer (BLL), and the presentation layer.

The DAL would include a database, such as SQL Server, that can contain tables and stored procedures. The DAL would also optionally include a component that would serve as a way to communicate with the database. This way, all calls to the database would take place through this component, rather than making queries directly to the database itself.

The BLL would be made up of a component that would store all business logic. Typically, this layer would be called from the presentation layer for all requests. The BLL would then make calls to the database access component whenever it needed access to data, or it would make calls directly to the database itself.

The presentation layer represents the user interface of the application. In Web application development, this might include the Active Server Pages (ASP) and Hypertext Transfer Language (HTML), or in the Microsoft .NET framework, it might include ASP.NET Web forms and Win forms.

Windows DNA and Microsoft .NET

One mistake that many developers make when they start working with the Microsoft .NET framework is thinking that the concepts outlined in Windows DNA no longer apply. This could not be further from the truth. Developers should still apply many of the Windows DNA concepts when designing their Microsoft .NET applications.

Components and Microsoft .NET

In the Microsoft .NET Framework, components are referred to as *assemblies*, and they are a fundamental part of the Microsoft .NET Framework, providing a number of objects that define the basic unit of development. This section will discuss how components in Microsoft .NET are structured and how you can build your own Microsoft .NET components.

Namespaces

Microsoft .NET components are organized into namespaces. All classes and namespaces in the Microsoft .NET framework are contained in either the `System` namespace or the `Microsoft` namespace. Each namespace contains additional namespaces or one or more classes which represent the objects in the framework. The following is an example of an object in that it is organized within two namespaces:

`System.Data.SqlClient`

`SqlClient` is the class name of this object, and the object is contained within the `Data` namespace, which is within the `System` namespace.

NOTE

Two classes can have the same name, as long as they are contained in different namespaces.

Declaring Namespaces

You declare namespaces by using the `namespace` keyword followed by the name of the namespace. You then place the namespace's contents within the beginning and end of the namespace declaration. An example of declaring a namespace in C# is shown in Listing 14.1 and an example of declaring a namespace in Visual Basic.NET is shown in Listing 14.2.

Listing 14.1 Declaring a Namespace (C#)

```
namespace n1 {  
    ...  
}
```

Listing 14.2 Declaring a Namespace (Visual Basic.NET)

```
Namespace n1  
    ...  
End Namespace
```

As mentioned in the previous section, namespaces can contain either classes or other namespaces. Namespaces that are contained within other namespaces are known as *nested namespaces*. There are two ways to declare a nested namespace. These methods are shown in Listing 14.3 and Listing 14.4.

Listing 14.3 Declaring a Nested Namespaces (C#)

First Way:

```
namespace n1 {  
    namespace n2 {  
        ...  
    }  
}
```

Second Way:

```
namespace n1.n2 {  
    ...  
}
```

Listing 14.4 Declaring a Nested Namespaces (Visual Basic.NET)

First Way:

```
Namespace n1  
    Namespace n2  
        ...  
    End Namespace  
End Namespace
```

Listing 14.4 continued

Second Way:

```
Namespace n1.n2
...
End Namespace
```

Classes

A major part of object-oriented programming is the concept of Classes. A class is a data structure that describes an object or a data member. This section will discuss classes within the Microsoft .NET Framework. We will first look at the different member types that are available to classes. Next, we will look at static versus dynamic members as well as member accessibility.

Class Members

In object-oriented programming, classes can contain both fields and methods. The Microsoft .NET framework provides two additional member types: properties and events. These types are discussed in the following sections and are summarized in Table 14.1.

Table 14.1 Class Member Types

Member Types	Description
Field	Represents a variable associated with the class
Method	Performs an action or task performed by the class
Property	Provides access to an attribute of a class
Event	Enables classes to provide notifications of an event

Fields

A field represents a variable that is associated with the class. As an example, consider an employee. We can associate a name and a phone number to any given employee. In terms of a class, the employee is the class and it would contain two fields or variables, name and phone. An example of a field is shown in Listing 14.5 and 14.6.

Listing 14.5 An Example of a Field (C#)

```
class Employee
{
    string Name;
    string Phone;
}
```

Listing 14.6 An Example of a Field (Visual Basic.NET)

```
Class Employee

    Dim Name As String
    Dim Phone As String

End Class
```

Methods

A method performs an action or a computation that is performed by the class. Methods are similar to functions or subroutines. Considering the employee object above, one action that you would perform for a given employee object, would be to print an address label. The action of printing the address label would represent a method for the *employee* class. An example is shown in Listing 14.7 and 14.8.

Listing 14.7 An Example of a Method (C#)

```
class Employee
{
    string Name;
    string Phone;

    void PrintAddressLabel() {
        //code here
    }
}
```

Listing 14.8 An Example of a Method (Visual Basic.NET)

```
Class Employee

    Dim Name As String
    Dim Phone As String

    Public Function PrintAddressLabel()
        'code here
    End Function

End Class
```

Properties

Properties are a natural extension of fields. A property provides access to an attribute of a class. Both fields and properties are named members that have associated types. However, properties simply provide access to the type, whereas fields also denote a storage location for the type.

A property provides access to its associated type to perform read and write operations. When you define a property, you can choose what operations to expose to client applications. The read operation is defined with the *get* keyword, and the write operation is defined with the *set* keyword. An example of a property that has both *get* and *set* implemented is shown in Listings 14.9 and 14.10.

Listing 14.9 An Example of a Property (C#)

```
class Employee
{
    string _name;
    public string Name
    {
        get {
            return _name;
        }
        set {
            _name = value;
        }
    }
}
```

Listing 14.10 An Example of a Property (Visual Basic.NET)

```
Class Employee

    Dim _Name As String

    Public Property Name() As String
        Get
            Return _Name
        End Get
        Set
            _Name = Value
        End Set
    End Property

End Class
```

Events

An event enables a class to provide notifications of an event within the class. Client applications can define an event handler that will execute when the class triggers a given event.

Static (Shared) and Instance Members

Members can be either instance members or static members (referred to as *shared members* in Visual Basic.NET); which one a member is affects the way it is called. Instance members require that an instance of the class be instantiated before the member can be called. Static members are shared across all instances of the class.

By default, all members are instance members, and you mark them as static by using the `static` keyword in C# and the `shared` keyword in Visual Basic.NET. An example of declaring static and instance members is shown in Listings 14.11 and 14.12. An example of referencing static and instance members is shown in Listings 14.13 and 14.14.

Listing 14.11 Declaring Static and Instance Members (C#)

```
using System;

namespace PureASP {

    class StaticInstance {

        public static long AddIntStatic(long x, long y) {
            return (x + y);
        }

        public long AddIntInstance(long x, long y) {
            return (x + y);
        }
    }
}
```

Listing 14.12 Declaring Static and Instance Members (Visual Basic.NET)

```
Imports System

Namespace PureASP {

    Public Class StaticInstance

        Public Shared Function AddIntStatic(x As Long, y As Long) As Long
            AddIntStatic = x + y
        End Function

        Public Function AddIntInstance(x As Long, y As Long) As Long
            AddIntInstance = x + y
        End Function

    End Class
}

End Namespace
```

Listing 14.13 Referencing Static and Instance Members (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="PureASP" %>

<script runat="server">

    void Page_Load(Object src, EventArgs e) {

        //Referencing the Static Method
        result1.Text = StaticInstance.AddIntStatic(3,2).ToString();
    }
</script>
```

Listing 14.13 continued

```
//Referencing the Instance Method
StaticInstance obj = new StaticInstance();
result2.Text = obj.AddIntInstance(4,3).ToString();

}

</script>

<html>

<head>
    <title>Referencing Static and Instance Members Example</title>
</head>

<body>
    Result from the AddInt Static Method:
    <asp:Label runat="server" id="result1" />
    <br>
    Result from the AddInt Instance Method:
    <asp:Label runat="server" id="result2" />
</body>

</html>
```

Listing 14.14 Referencing Static and Instance Members (Visual Basic.NET)

```
<% @Page Language="VB" %>
<% @Import Namespace="PureASP" %>

<script runat="server">

Protected Sub Page_Load(Src As Object, E As EventArgs)

    'Referencing the Static Method
    result1.Text = StaticInstance.AddIntStatic(3,2).ToString()

    'Referencing the Instance Method
    Dim obj As StaticInstance = new StaticInstance()
    result2.Text = obj.AddIntInstance(4,3).ToString()

End Sub

</script>

<html>
```

Listing 14.14 continued

```
<head>
    <title>Referencing Static and Instance Members Example</title>
</head>

<body>
    Result from the AddInt Static Method:
    <asp:Label runat="server" id="result1" />
    <br>
    Result from the AddInt Instance Method:
    <asp:Label runat="server" id="result2" />
</body>

</html>
```

Member Accessibility

One of the key object-oriented programming concepts associated with components is encapsulation. With encapsulation, the user can access only necessary properties and methods; a number of other methods and properties are used internally and the client cannot access them. In Microsoft .NET, a member can be marked as public, private, protected, or internal. Table 14.2 summarizes these accessibility settings.

Table 14.2 Member Accessibility Settings

Access Modifier	Description
Public	The member is publicly available.
Private	The member is available only to the class itself.
Protected	The member is available only to the class itself and to objects that are derived from the class.
Internal	The member is available only to objects within the same assembly as the class.

Compiling a Microsoft .NET Component

If you are developing components with an editor that doesn't have built-in compiling (such as Visual Studio.NET), you will have to compile components manually. You do this by using the command-line compilers included with the Microsoft .NET framework.

The Visual Basic.NET command-line compiler is a file called **vbc.exe**, and the C# compiler is a file called **csc.exe**. To get a list of all the compiler options, type **vbc** or **csc** at a command prompt. Table 14.3 summarizes the key compiler options you will need in order to compile your components.

**NOTE**

When Microsoft .NET is installed, the system PATH is updated to include the Microsoft .NET installation directory. Therefore you can run vbc.exe and csc.exe from anywhere in the file system.

Table 14.3 Command-Line Compiler Options

Compiler Option	Description
/out:<file>	Specifies the filename of the compilation output
/target:<type>	Specifies the target type (short form: /t), which can be one of the following: <ul style="list-style-type: none">• library (A library assembly, component, or DLL)• exe (A console application)• winexe (A Windows application)• module (A module to add to an assembly)
/reference:<file_list>	References metadata from the specified assembly (short form: /r)

The following is an example of a C# command-line compile statement for a library (that is, a dynamic link library [DLL]) that uses the System.Data namespace:

```
csc /out:bin/mycomponent.dll /t:library /r:System.Data.dll
```

The same library written in Visual Basic.NET would look like this:

```
vbc /out:bin/mycomponent.dll /t:library /r:System.Data.dll
```

Accessing Components from ASP.NET Applications

After you compile a component, you need to place it in the bin folder located in the Web application's root directory. From this directory, the component is available to the Web application.

Let's look at an example of accessing a Microsoft .NET component (that is, assembly) from an ASP.NET Web form. Listing 14.15 and Listing 14.16 show the referenced component. Listing 14.17 and Listing 14.18 demonstrate how to access this component from an ASP.NET Web form.

Listing 14.15 An Example of a Microsoft .NET Component (C#)

```
using System;
```

```
namespace PureASP {
```

```
    public class Sample {
```

Listing 14.15 continued

```
public long AddInt(long x, long y) {  
    return (x + y);  
}  
  
}  
}
```

Listing 14.16 An Example of a Microsoft .NET Component (Visual Basic.NET)

```
Imports System
```

```
Namespace PureASP
```

```
Public Class Sample  
  
    Public Function AddInt(x As Long, y As Long) As Long  
        AddInt = x + y  
    End Function  
  
End Class
```

```
End Namespace
```

Listing 14.17 Accessing an Component from an ASP.NET Web Form (C#)

```
<% @Page Language="C#" %>  
<% @Import Namespace="PureASP" %>  
  
<script runat="server">  
  
    void Page_Load(Object src, EventArgs e) {  
  
        Sample obj = new Sample();  
        result.Text = obj.AddInt(4,3).ToString();  
  
    }  
  
</script>  
  
<html>  
    <head>  
        <title>Accessing Component Example</title>  
    </head>
```

Listing 14.17 continued

```
<body>
    Result from the AddInt Method: <asp:Label runat="server" id="result" />
</body>

</html>
```

***Listing 14.18 Accessing an Component from an ASP.NET Web Form
(Visual Basic.NET)***

```
<% @Page Language="VB" %>
<% @Import Namespace="PureASP" %>

<script runat="server">

    Protected Sub Page_Load(Src As Object, E As EventArgs)
        Dim obj As Sample = new Sample()
        result.Text = obj.AddInt(4,3).ToString()

    End Sub

</script>

<html>
    <head>
        <title>Accessing Component Example</title>
    </head>

    <body>
        Result from the AddInt Method: <asp:Label runat="server" id="result" />
    </body>
</html>
```

COM Interoperability

You will often use Microsoft .NET to develop new applications and components, but you will also need to use Microsoft .NET to utilize and support existing applications and libraries. For optimum performance, you would want to upgrade all existing applications and libraries to Microsoft .NET, but this would simply not be practical in all cases. Luckily, the Microsoft .NET framework contains functionality that enables interoperability with existing COM components. This section discusses the options provided for referencing existing components. There are two types of references that you can make to existing COM components. These are known as late-bound and early-bound references. *Late-bound* refers to objects that are created and set to the object

type at runtime, and *early-bound* are objects that are created and set to the object type at design time. The following sections will discuss how to create late-bound and early-bound references to existing COM components.

Late-Bound COM References

Microsoft .NET offers immediate interoperability with COM components by providing a way for developers to late-bind to existing COM components. You do this by using the `Microsoft .NET Server.CreateObject()` method, which is similar to the `Server.CreateObject()` method provided in COM. An example of using this method is shown in Listing 14.19 and 14.20.

Listing 14.19 A Late-Bound COM Reference (C#)

```
Object myObject;  
myObject = Server.CreateObject("PureASP.Interop")
```

Listing 14.20 A Late-Bound COM Reference (Visual Basic.NET)

```
Dim MyObject As Object  
MyObject = Server.CreateObject("PureASP.Interop ")
```

Early-Bound COM References

You can also reference early-bound COM components from Microsoft .NET. You do this by using a built-in utility that creates a proxy class. Proxy classes provide the necessary plumbing to access the COM component from the Microsoft .NET framework.

The following is an example of creating a proxy class that allows us to reference an early-bound COM component written in Visual Basic 6.0. The COM component has one simple method, `AddInt`, that accepts two numbers and returns the sum of the numbers. The COM component is shown in Listing 14.21.

Listing 14.21 A Visual Basic 6.0 COM Component

```
Public Function AddInt(ByVal x As Long, ByVal y As Long) As Long  
    AddInt = x + y  
End Function
```

You can compile this component by using Visual Basic 6.0 into a COM library with the filename `PureASP.dll`. After the component is compiled and registered, you can use the Type Library Importer utility that is included with the Microsoft .NET framework (as a file named `tlbimp.exe`). The following command creates a proxy class named `PureASP_proxy.dll`:

```
tlbimp PureASP.dll /out:bin\PureASP_proxy.dll
```

The example shown in Listing 14.22 and 14.23 demonstrate how to use the COM Visual Basic 6.0 COM component from ASP.NET via an early-bound proxy class.

Listing 14.22 Using the COM Proxy Class from ASP.NET (C#)

```
<% @Page Language="C#" %>
<% @Import Namespace="PureASP_proxy" %>

<script runat="server">
    void Page_Load(Object src, EventArgs e) {
        Interop pureInterop = new Interop();
        result.Text = pureInterop.AddInt(2,3).ToString();
    }
</script>

<html>
    <head>
        <title>Interop Example</title>
    </head>

    <body>
        Result from the AddInt method: <asp:Label runat="server" id="result" />
    </body>
</html>
```

Listing 14.23 Using the COM Proxy Class from ASP.NET (Visual Basic.NET)

```
<% @Page Language="VB" %>
<% @Import Namespace="PureASP_proxy" %>

<script runat="server">

Protected Sub Page_Load(Src As Object, E As EventArgs)

    Dim PureInterop As Interop = new Interop()
    result.Text = PureInterop.AddInt(2,3).ToString()

End Sub

</script>

<html>
    <head>
        <title>Interop Example</title>
    </head>
```

Listing 14.23 continued

```
<body>
    Result from the AddInt method: <asp:Label runat="server" id="result" />
</body>

</html>
```

COM+ Services: Using Transactions from Microsoft .NET

COM+ offers a number of services that can add value to Microsoft .NET applications. Some of these services include Microsoft Message Queuing (MSMQ), Microsoft Transaction Processing, and Connection Pooling. You can access these services from Microsoft .NET applications through the `System.EnterpriseServices` namespace.

One of the most popular COM+ services is Microsoft Transaction Processing. A Microsoft .NET component that supports COM+ transactions is considered a serviced component. A *serviced component* is a class that derives from the `System.EnterpriseServices.ServicedComponent` class. Listing 14.24 and Listing 14.25 show the structure for a basic serviced component.

Listing 14.24 A Transaction Skeleton (C#)

```
[assembly: ApplicationName("PureASP Trans")]
[assembly: AssemblyKeyFileAttribute("PureASPTrans.snk")]

namespace PureASP
{
    [Transaction(TransactionOption.Required)]
    public class Trans:ServicedComponent
    {
        // code here
    }
}
```

Listing 14.25 A Transaction Skeleton (Visual Basic.NET)

```
<assembly: ApplicationName("PureASP Trans")>
<assembly: AssemblyKeyFileAttribute("PureASPTrans.snk")>

Namespace PureASP

    Public Class <Transaction(TransactionOption.Required)> Trans
        Inherits ServicedComponent
        ' code here
    End Class

End Namespace
```

Notice Listings 14.24 and 14.25 differ from a standard class structure in three ways. First, the class implements `ServicedComponent`, as discussed previously. Second, the `Transaction` attribute is placed on the class that specifies that the class is going to participate in a transaction. The `Transaction` attribute takes one argument, of type `TransactionOption`. In the case of the skeletons in Listings 14.24 and 14.25, the `Transaction` attribute is marked as required, which means that if a transaction exists, it will be shared, and if a transaction does not exist, a new one will be created. All the values for `TransactionOption` are summarized in Table 14.4.

Table 14.4 TransactionOption Values

Values	Description
<code>Disabled</code>	Ignore any transaction in the current context.
<code>NotSupported</code>	Create the component in a context with no governing transaction.
<code>Required</code>	Share a transaction if one exists; create a new transaction if necessary.
<code>RequiresNew</code>	Create the component with a new transaction, regardless of the state of the current context.
<code>Supported</code>	Share a transaction if one exists.

Third, there are two assembly attributes before the namespace declaration. The first of these attributes (`ApplicationName`) assigns the name of the COM+ application. The second (`AssemblyKeyFileAttribute`) signs the component with a strong name, which is discussed in the following section.

Committing a Transaction

When you execute a method that participates in a transaction, any action that takes place before the transaction is completed has the potential to be rolled back. To commit a transaction, you use the `SetComplete` method of the `ContextUtil` class. In C# this statement would look like this:

```
ContextUtil.SetComplete();
```

And in Visual Basic.NET it would look like this:

```
ContextUtil.SetComplete()
```

Aborting a Transaction

If an application causes an exception to occur before the transaction is committed, all the actions that participated in the transaction would be rolled back and put back into the state they were in before the transaction began. To abort a transaction, you use the `SetAbort` method of the `ContextUtil` class. In C# this statement would look like this:

```
ContextUtil.SetAbort();
```

And in Visual Basic.NET it would look like this:

```
ContextUtil.SetAbort()
```

Using AutoComplete

If you do not want to manually commit and abort your transactions, you can use AutoComplete, which handles the committing and aborting of the transaction automatically. The way it works is simple: If the transaction executes without any exceptions, the transaction is committed; if any exceptions are thrown while processing the transaction, the transaction is automatically aborted.

To use AutoComplete, add the AutoComplete attribute to the methods you want to be handled automatically. An example of AutoComplete is shown in the following section.

NOTE

When you use AutoComplete, you can still control when a transaction is aborted by throwing exceptions in your application. Some consider this a better development technique than aborting transactions by using `ContextUtil`.

A Transaction Example

This section further explains how to use COM+ transactions in Microsoft .NET applications by walking through an example. This example allows users to transfer funds from one credit card to another. Two business rules determine whether the transfer will succeed: An account balance cannot be less than zero, and an account balance cannot be more than a set limit. The example is shown in Figure 14.1.

The screenshot shows a Microsoft Internet Explorer window with the title bar 'COM+ Transactions Example - Microsoft Internet Explorer'. The address bar contains the URL 'http://morpheus/transactions/accounts.aspx'. The main content area has a heading 'COM+ Transactions Example'.

Table of Account Balances:

AccountName	Balance	Limit
VISA	2000	4000
MASTERCARD	2200	3500
DISCOVER	1400	7500
AMERICAN EXPRESS	2400	3000

Transfer Amount (Integer):

Transfer From: VISA MASTERCARD DISCOVER AMERICAN EXPRESS

Transfer To: VISA MASTERCARD DISCOVER AMERICAN EXPRESS

Execute Transaction

Figure 14.1

A Sample Application that Supports Transactions

The first step in creating this example is to create the database table and the stored procedures. The stored procedure used to debit an account is shown in Listing 14.26.

Listing 14.26 The AccountsDebit Stored Procedure

```
CREATE PROCEDURE AccountsDebit
(
    @AccountId int,
    @Amount money
)
AS

DECLARE @Test_Success int
DECLARE @Test_Balance int

SELECT
    @Test_Success = Count(AccountId)
FROM
    Accounts
WHERE
    AccountId = @AccountId

IF (@test_Success < 1)
    RAISERROR('The source account ID, %d, is invalid.', 16, 1, @AccountId)

SELECT
    @Test_Balance = Balance
FROM
    Accounts
WHERE
    AccountId = @AccountId

IF (@Test_Balance < @Amount)
    RAISERROR('Debit amount is greater than balance', 16, 1)
ELSE
BEGIN

    UPDATE
        Accounts
    SET
        Balance = Balance - @Amount
    WHERE
        AccountId = @AccountId

END
GO
```

The next step in is to create the component that will represent the Accounts object. This object will contain one method called ExecuteTransaction that will require a

transaction. We will use AutoComplete to automatically handle committing and aborting the transaction. The component's source is shown in Listing 14.27.

Listing 14.27 Accounts Component Supporting Transactions (C#)

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.EnterpriseServices;
using System.Reflection;

[assembly: ApplicationName("Accounts")]
[assembly: AssemblyKeyFileAttribute("Accounts.snk")]

namespace PureASP
{

    [Transaction(TransactionOption.Required)]
    public class Accounts:ServicedComponent
    {

        [AutoComplete(true)]
        public bool ExecuteTransaction(
            int amount, int sourceAccountId, int targetAccountId)
        {
            Exception eSourceTargetIdentical =
                new Exception("Source and target accounts cannot be the Same");

            if (sourceAccountId == targetAccountId)
                throw (eSourceTargetIdentical);

            //debit the source account
            DebitAccount(sourceAccountId, amount);

            //credit the target account
            CreditAccount(targetAccountId, amount);

            return true;
        }

        public static SqlDataReader ListAccounts()
        {

            SqlConnection conn = new SqlConnection();
            SqlCommand cmd = new SqlCommand();

            conn.ConnectionString = "server=localhost;uid=sa;pwd=;database=accounts";
```

Listing 14.27 continued

```
cmd.CommandText = "AccountsList";
cmd.CommandType = CommandType.StoredProcedure;
cmd.Connection = conn;

conn.Open();
SqlDataReader result =
    cmd.ExecuteReader(CommandBehavior.CloseConnection);

return result;

}

public static SqlDataReader ListAccountDetails()
{

    SqlConnection conn = new SqlConnection();
    SqlCommand cmd = new SqlCommand();

    conn.ConnectionString = "server=localhost;uid=sa;pwd=;database=accounts";

    cmd.CommandText = "AccountsListDetails";
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Connection = conn;

    conn.Open();
    SqlDataReader result =
        cmd.ExecuteReader(CommandBehavior.CloseConnection);

    return result;

}

private void UpdateAccount(string command, int accountId, int amount)
{

    Exception eInvalidCommand = new Exception("Invalid Command");

    SqlConnection conn = new SqlConnection();
    SqlCommand cmd = new SqlCommand();

    conn.ConnectionString = "server=localhost;uid=sa;pwd=;database=accounts";

    if (command == "DEBIT")
        cmd.CommandText = "AccountsDebit";

    else if (command == "CREDIT")
        cmd.CommandText = "AccountsCredit";
```

Listing 14.27 continued

```
else
    throw(eInvalidCommand);

cmd.CommandType = CommandType.StoredProcedure;
cmd.Connection = conn;

// add parameters
SqlParameter pAccId = new SqlParameter("@AccountId", SqlDbType.Int, 4);
pAccId.Value = accountId;
cmd.Parameters.Add(pAccId);

// add parameters
SqlParameter pAmt = new SqlParameter("@Amount", SqlDbType.Int, 4);
pAmt.Value = amount;
cmd.Parameters.Add(pAmt);

conn.Open();
cmd.ExecuteNonQuery();

}

protected void CreditAccount(int accountId, int amount)
{
    UpdateAccount("CREDIT", accountId, amount);
}

protected void DebitAccount(int accountId, int amount)
{
    UpdateAccount("DEBIT", accountId, amount);
}

}
```

For the component to participate in a transaction, it must be signed with a strong name. You do this by first creating a cryptographic key pair, using the Strong Name Utility (sn.exe) included with the Microsoft .NET framework. The key pair used for the Accounts object transaction example is generated by the Strong Name Utility by executing the following statement:

```
sn -k Accounts.snk
```

The key pair contains the assembly's name and version number. In addition, it has a public key and a digital signature.

After you have signed a strong name to the component, you compile the component by using the following statement:

```
csc /out:bin\Accounts.dll /r:System.EnterpriseServices.dll  
/r:System.Data.dll /target:library Accounts.cs
```

After the component is compiled, you can reference it from the Web form, which is shown in Listing 14.28.

Listing 14.28 The Accounts Object Web Form (C#)

```
<% @Page Language="C#" debug="true"%>  
<% @Import Namespace="PureASP" %>  
  
<script runat="server">  
  
void Page_Load(Object src, EventArgs e) {  
  
    if (!Page.IsPostBack) DisplayAccountSummary();  
  
}  
  
void DisplayAccountSummary() {  
  
    SourceAccount.DataSource = Accounts.ListAccounts();  
    TargetAccount.DataSource = Accounts.ListAccounts();  
    AccountSummary.DataSource = Accounts.ListAccountDetails();  
  
    Page.DataBind();  
  
}  
  
void ExecuteTransaction(Object src, EventArgs e) {  
  
    if (Page.IsValid) {  
  
        int amt = Int32.Parse(Amount.Text);  
        int tgtAcct = Int32.Parse(TargetAccount.SelectedItem.Value);  
        int srcAcct = Int32.Parse(SourceAccount.SelectedItem.Value);  
  
        PureASP.Accounts acc = new PureASP.Accounts();  
        acc.ExecuteTransaction(amt, srcAcct, tgtAcct);  
  
        DisplayAccountSummary();  
    }  
}
```

Listing 14.28 continued

```
Result.Text = "Successfully Transferred $" + Amount.Text;// +
// "from " + SourceAccount.SelectedItem.Text + " to " +
// TargetAccount.SelectedItem.Text;

}

}

</script>

<html>
<head>
    <title>COM+ Transactions Example</title>
</head>
<body>
    <form id="frm" method="post" runat="server">
        <p>
            <asp:Label id="Label3" runat="server" Font-Bold="True" Font-Size="14pt"
                Font-Names="Verdana">COM+ Transactions Example</asp:Label>
        </p>
        <p>
            <asp:DataGrid runat="server" width="300" id="AccountSummary"
                cellpadding="3">
                <HeaderStyle Font-Size="8pt" Font-Names="Verdana"
                    Font-Bold="True" ForeColor="Black" BackColor="Beige" />
                <ItemStyle Font-Size="8pt" Font-Names="Verdana" />
            </asp:DataGrid>
        </p>
        <p>
            <asp:Label id="Label4" runat="server" Font-Bold="True"
                Font-Size="10pt" Font-Names="Verdana">Transfer Amount (Integer):</asp:Label>
            &nbsp;
            <asp:TextBox id="Amount" runat="server" />
        </p>
        <table cellspacing="0" cellPadding="5" width="400" border="0">
            <tr>
                <td width="200">
                    <asp:Label id="Label1" runat="server" Font-Bold="True"
                        Font-Size="10pt" Font-Names="Verdana" Text="Transfer From: " />
                </td>
                <td width="200">
                    <asp:Label id="Label2" runat="server" Font-Bold="True"
                        Font-Size="10pt" Font-Names="Verdana" Text="Transfer To: " />
                </td>
            </tr>
            <tr>
```

Listing 14.28 continued

```
<td width="200">
    <asp:RadioButtonList id="SourceAccount" runat="server"
        Font-Size="8pt" Font-Names="Verdana" DataTextField="AccountName"
        DataValueField="AccountId" />
</td>
<td width="200">
    <asp:RadioButtonList id="TargetAccount" runat="server"
        Font-Size="8pt" Font-Names="Verdana" DataTextField="AccountName"
        DataValueField="AccountId" />
</td>
</tr>
</table>
<p>
    <asp:Button id="Button1" runat="server" Font-Bold="True"
        Text="Execute Transaction" BackColor="Beige"
        OnClick="ExecuteTransaction" />
</p>
<asp:Label runat="server" id="Result" Font-Size="10pt"
    Font-Names="Verdana" Forecolor="red"/>
</form>
</body>
</html>
```

The final step is to provide access to the `System.EnterpriseServices` namespace from the Web form. You do this by adding an entry in the `<assemblies>` section in the `web.config` file for the application. The `web.config` file for this example is shown in Listing 14.29.

Listing 14.29 The Accounts Object web.config File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

    <system.web>
        <compilation>

            <assemblies>
                <add assembly="System.EnterpriseServices" />
            </assemblies>

        </compilation>
    </system.web>

</configuration>
```


CHAPTER 15

Building Web Services

This chapter details how to create and use a Web service. The steps in creating a Web service are as follows:

1. Design the Web service interface.
2. Develop the Web service.
3. Enable programmatic discovery of the Web service (optional).
4. Use `WSDL.exe` to build a proxy class in the consuming application.
5. Compile the consuming application with the proxy class.

Each of these steps is detailed in the following sections. For an overview of Web services, please see Chapter 8, “Overview of Web Services.”

Creating a Web Service

Creating a Web service is a straightforward process. As when creating any other program, you should plan the interface before you start coding. Implementing the details of the Web service in code is as easy as creating a Web form.

Designing a Web Service Interface

The most important part of designing a Web service is deciding on exactly which functionality to expose. The public interface should be powerful enough to be useful yet generic enough to be reusable. The section “Exposing Useful Functionality,” later in this chapter, shows an example of a powerful and reusable Web service.

Developing a Web Service

Developing a Web service is straightforward. The first step is to create a file with the extension `.asmx` anywhere in the virtual directory of your .NET application. The Microsoft .NET

framework automatically creates the framework needed for a Web service by recognizing the .asmx extension. The Web service in Listings 15.1 and 15.2 is a very simple anonymous remailer that accepts four arguments, sends an e-mail message, and then returns nothing to the caller. In this example, the name of the file is `Remailer.asmx`.

Listing 15.1 A Very Simple Web Service (C#)

```
<%@ WebService Language="C#" Class="SendMail"%>

using System.Web.Services;
using System.Web.Mail;

public class SendMail : WebService {

    [ WebMethod ]
    public void Send(string from, string to,
                     string subject, string messageText) {

        System.Web.Mail.SmtpMail.Send(from, to, subject, messageText);
    }
}
```

Listing 15.2 A Very Simple Web Service (Visual Basic.NET)

```
<%@ WebService Language="VB" Class="SendMail"%>

imports System.Web.Services
imports System.Web.Mail

Public Class SendMail public sub <WebMethod ()> Send(from as string,
                                                    mailTo as string, subject as string, messageText as string)
    System.Web.Mail.SmtpMail.Send(from, mailTo, subject, messageText)
End sub
End Class
```

The first line of any Web service contains the `WebService` directive, as in the following example:

```
<%@ WebService Language="VB" Class="SendMail"%>
```

This directive contains two settings for the Web service. The `Class` attribute is the name of the Web service class. This can be any managed class in the application. In other words, you can use a precompiled Web service, as discussed later in this chapter, in the section “Using Precompiled Web Services.”

The `Language` attribute defines the programming language that the Web service was created in. If it is not specified, the default is C#. Web services can be created in any language that is supported by the .NET framework.

The next few lines add the `System.Web.Services` and `System.Web.Mail` namespaces to the Web service. Only `System.Web.Services` is required to get a web service

operational. The `System.Web.Mail` namespace is added so that the `SmtpMail` object can be used in the service.

For now, the Web service is being defined inline in the `.asmx` file, so the next line defines the `SendMail` Web service class, which derives from the `WebService` base class.

All that is left is to actually declare the public methods of the Web service. This is done differently depending on the language you are working with. In C#, the `WebMethod` attribute is placed before the declaration of the method:

```
[ WebMethod ]
public void Send(string from, string to, string subject, string messageText)
```

In Visual Basic.NET, the `WebMethod` attribute is placed inside the function call:

```
public sub <WebMethod ()> Send(from as string, _
                               mailTo as string, subject as string, messageText as
                               string)
```

The contents of the method are standard code you would find in any Web form or component.

Building Web Services into an Existing Application

One popular type of Web application is shopping-cart-based e-commerce sites such as Amazon.com. You are likely to have used many of these sites at some point. Some sites, such as Amazon.com, have excellent order-tracking systems in place. Unfortunately, the majority of e-commerce sites do not have good order-tracking systems.

When developing an e-commerce site in ASP.NET, you can use Web services to offer a standardized way of checking the status of an order from any remote client. Microsoft developed a Web service to perform exactly this task for its fictitious online e-commerce site, located at www.IBuySpy.com. Listing 15.3 contains the code for this Web service. IBuySpy.com first verifies the user's credentials, which are passed in as arguments, and then it retrieves information based on the order ID.

Listing 15.3 The C# Code of the IBuySpy Web Service

```
*****  
//  
// InstantOrder.CheckStatus() Method  
//  
// The CheckStatus method enables a remote client to programmatically  
// query the current status of an order in the IBuySpy System.  
//  
*****  
[WebMethod(Description="The CheckStatus method enables a remote client
```

Listing 15.3 continued

```
        to programmatically query the current status
        of an order in the IBuySpy System.", EnableSession=false)]
public OrderDetails CheckStatus(string userName,
                                string password, int orderID) {

    // Login client using provided username and password
    IBuySpy.CustomersDB accountSystem = new IBuySpy.CustomersDB();
    String customerId = accountSystem.Login(userName, password);

    if (customerId == null) {
        throw new Exception("Error: Invalid Login!");
    }

    // Return OrderDetails Status for Specified Order
    IBuySpy.OrdersDB orderSystem = new IBuySpy.OrdersDB();
    return orderSystem.GetOrderDetails(orderID);
}
}
```

Listing 15.4 The Visual Basic.NET Code of the IBuySpy Web Service

```
' ****
'
' InstantOrder.CheckStatus() Method
'
' The CheckStatus method enables a remote client to programmatically
' query the current status of an order in the IBuySpy System.
'
' ****

Public Function <WebMethod()> CheckStatus(userName As String, _
                                             password As String, OrderID As Integer) As OrderDetails

    ' Login client using provided username and password
    Dim accountSystem As IBuySpy.CustomersDB = New IBuySpy.CustomersDB()
    Dim customerId As String = accountSystem.Login(userName, password)

    If customerId = Nothing Then
        throw new Exception("Error: Invalid Login!")
    End If

    ' Return OrderDetails Status for Specified Order
    Dim orderSystem As IBuySpy.OrdersDB = New IBuySpy.OrdersDB()
    Return orderSystem.GetOrderDetails(orderID)

End Function
```

Exposing Useful Functionality

In addition to the Web service in Listings 15.7 and 15.8, which enables any remote client or application to check the status of an order, several other useful Web services could greatly improve interoperability between sites. For instance, say that a fictitious partner site to IBuySpy, named SpyGear, emerged. Rather than compete directly with IBuySpy, SpyGear would like to offer another storefront for IBuySpy—that is, another completely different branded site where identical merchandise can be purchased.

One easy way to achieve this goal would be to create a list of private Web services. A suite of Web services could be used to enumerate the IBuySpy catalog to a remote client and get more information about specific products. Additionally, a suite of order-based Web services could be created to enable a remote application to place orders directly into the IBuySpy system, to avoid having two separate databases.

In fact, IBuySpy already contains a Web service that will place an order into the IBuySpy database (see Listing 15.5).

Listing 15.5 Placing an Order on IBuySpy, Using Web Services, in C#

```
//*****
//  
// InstantOrder.OrderItem() Method  
//  
// The OrderItem method enables a remote client to programmatically  
// place an order using a webservice.  
//  
//*****  
  
[WebMethod(Description="The OrderItem method enables a remote  
client to programmatically place an order  
using a WebService.", EnableSession=false)]  
public OrderDetails OrderItem(string userName, string password,  
int productID, int quantity) {  
  
    // Login client using provided username and password  
    IBuySpy.CustomersDB accountSystem = new IBuySpy.CustomersDB();  
    String customerId = accountSystem.Login(userName, password);  
  
    if (customerId == null) {  
        throw new Exception("Error: Invalid Login!");  
    }  
  
    // Add Item to Shopping Cart  
    IBuySpy.ShoppingCartDB myShoppingCart = new IBuySpy.ShoppingCartDB();  
    myShoppingCart.AddItem(customerId, productID, quantity);  
  
    // Place Order
```

Listing 15.5 continued

```
IBuySpy.OrdersDB orderSystem = new IBuySpy.OrdersDB();
int orderID = orderSystem.PlaceOrder(customerId, customerId);

// Return OrderDetails
return orderSystem.GetOrderDetails(orderID);
}
```

Listing 15.6 Placing an Order on IBuySpy, Using Web Services, in Visual Basic.NET

```
' ****
'
' InstantOrder.OrderItem() Method
'
' The OrderItem method enables a remote client to programmatically
' place an order using a webservice.
'
' ****

Public Function <WebMethod()> OrderItem(userName As String, _
    password As String, productID As Integer, _
    quantity As Integer) As OrderDetails

    ' Login client using provided username and password
    Dim accountSystem As IBuySpy.CustomersDB = New IBuySpy.CustomersDB()
    Dim customerId As String = accountSystem.Login(userName, password)

    If customerId = Nothing Then
        Throw New Exception("Error: Invalid Login!")
    End If

    ' Add Item to Shopping Cart
    Dim myShoppingCart As IBuySpy.ShoppingCartDB = New IBuySpy.ShoppingCartDB()
    myShoppingCart.AddItem(customerId, productID, quantity)

    ' Place Order
    Dim orderSystem As IBuySpy.OrdersDB = New IBuySpy.OrdersDB()
    Dim orderID As Integer = orderSystem.PlaceOrder(customerId, customerId)

    ' Return OrderDetails
    Return orderSystem.GetOrderDetails(orderID)

End Function
```

Consuming Web Services

Before you are ready to use a Web service in your application, you must perform a few steps. First, the Web service itself must be located and interrogated. You can find

more information on the exposed Web service by navigating directly to the Web service URL. This is covered in the section “Analyzing the WSDL Contract” later in this chapter.

After discovering the location and methods that the Web service supports, the next step is to create a proxy class of the Web service. The proxy class is compiled into your application and enables you to program against the Web service interface. Proxy classes are covered in detail later in this chapter, in the section “Generating a Proxy Class.”

After the proxy class has been created, using the Web service is as simple as instantiating the class and then calling the Web service methods, as you would with any other method.

Analyzing the WSDL Contract

If you browse to an .asmx page in the Microsoft .NET framework, you see a page much like the one in Figure 15.1.

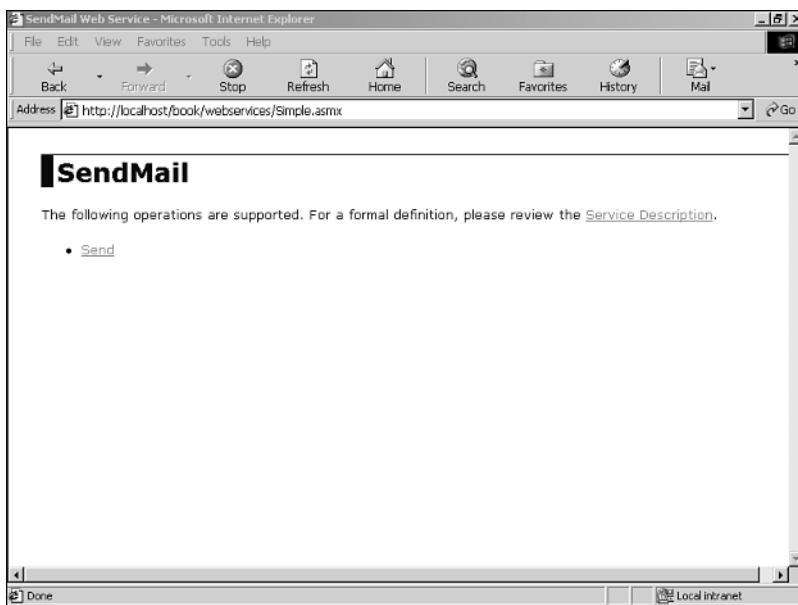
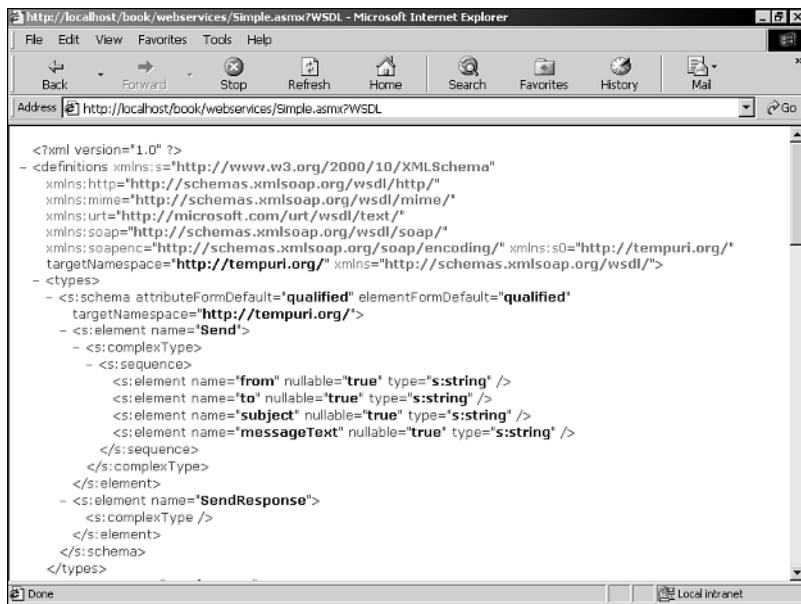


Figure 15.1

The SendMail.asmx Web service.

The name of the Web service class appears in Figure 15.1, followed by a link to the WSDL page, which looks very similar to Figure 15.2.

**Figure 15.2**

The SendMail *SDL* contract viewed in Microsoft Internet Explorer 5.5.

The *SDL* contract is an XML document that defines the various methods of the Web service and the arguments required. In addition, each supported transport type is detailed individually. For instance, Listing 15.7 is part of the XML code that describes the *Send* method of the *SendMail* Web service.

Listing 15.7 The WSDL Description of the *Send* Method

```
<s:element name="Send">
  <s:complexType>
    <s:sequence>
      <s:element name="from" nullable="true" type="s:string" />
      <s:element name="to" nullable="true" type="s:string" />
      <s:element name="subject" nullable="true" type="s:string" />
      <s:element name="messageText" nullable="true" type="s:string" />
    </s:sequence>
  </s:complexType>
</s:element>
```

Testing a Web Service

When navigating to a Web service in a browser, notice that all the public methods of the Web service are listed on the page. If you select the *Send* method of the *SendMail* Web service, you see a screen like the one in Figure 15.3.



Figure 15.3

The default SendMail Web service form.

Figure 15.3 shows a default form that enables a developer to quickly and easily test a Web service without having to create the code to consume the Web service. In addition, on the same page, you can view the actual SOAP, Hypertext Transfer Protocol (HTTP) GET, and HTTP POST requests and responses, which are shown in Listings 15.8 through 15.13. This provides easily accessible debugging information as it can make programmatic errors or lapses in judgment immediately obvious.

Listing 15.8 The SOAP SendMail Send Request

```
POST /book/WebServices/Simple.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset="utf-8"
Content-Length: length
SOAPAction: "http://tempuri.org/Send"

<?xml version="1.0"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/soap/envelope">
  <soap:Body>
    <Send xmlns="http://tempuri.org/">
      <from>string</from>
      <to>string</to>
```

Listing 15.8 continued

```
<subject>string</subject>
<messageText>string</messageText>
</Send>
</soap:Body>
</soap:Envelope>
```

Listing 15.9 The SOAP SendMail Send Response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: length
```

```
<?xml version="1.0"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/soap/envelope">
  <soap:Body>
    <SendResponse xmlns="http://tempuri.org/" />
  </soap:Body>
</soap:Envelope>
```

Listing 15.10 The HTTP GET SendMail Send Request

```
GET /book/WebServices/Simple.asmx/Send?from=string
     &to=string&subject=string&messageText=string HTTP/1.1
Host: localhost
```

Listing 15.11 The HTTP GET SendMail Send Response

```
HTTP/1.1 200 OK
```

Listing 15.12 The HTTP POST SendMail Send Request

```
POST /book/WebServices/Simple.asmx/Send HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length

from=string&to=string&subject=string&messageText=string
```

Listing 15.13 The HTTP POST Sendmail Send Response

```
HTTP/1.1 200 OK
```

Generating a Proxy Class

As mentioned earlier in this chapter, in order for an application to consume a Web service, a proxy class must first be created. A proxy class simply defines the public interfaces of a remote Web service so that you can reference them locally. It also

contains the necessary “plumbing” to connect to the remote Web service and, if required, return a result to your program. Remember that your application is not aware that the Web service methods are being performed remotely.

Theoretically, any of the compilers could be built to automatically analyze the remote Web services so that the proxy class would not be necessary. However, this would require an Internet connection at compile time as well as the creation of new syntax to directly call a remote Web service. A proxy class is used because it is the simplest method.

A utility is not required to create a proxy class. Anyone armed with a text editor, a target Web service WSDL contract, and enough free time could perform this task. However, there is absolutely no benefit to generating a proxy class by hand. Microsoft provides the utility `WSDL.exe`, which ships with the .NET framework, to analyze the target Web service WSDL contract and build a proper proxy class in only a few seconds.

You can find `WSDL.exe` in the main `/bin` directory of your .NET framework installation directory. This directory is automatically added to your system path during .NET installation so that you can call this program from anywhere in a DOS session or script.

The WSDL application is used in the following manner:

```
WSDL <options> <url or path> <url or path>
```

Table 15.1 describes the arguments to the WSDL executable.

Table 15.1 WSDL Arguments

Argument	Description
<code>/nologo</code>	Suppresses the banner.
<code>/language:<language></code>	Specifies the language to use for the generated proxy class. Can be <code>cs</code> , <code>vb</code> , <code>js</code> , or a fully qualified name for any .NET language.
<code>(/l)</code>	
<code>/server</code>	Generates an abstract class for a Web service implementation based on the contracts. The default is to generate client proxy classes.
<code>/namespace:<namespace></code>	Specifies the namespace for the generated proxy or template. The default namespace is the global namespace.
<code>(/n)</code>	
<code>/out:<filename></code>	Specifies the filename for the generated proxy code. The default name is derived from the service name.
<code>(/o)</code>	
<code>/nobackup</code>	Overwrites the existing output file. The default is to produce a <code>.bak</code> file.
<code>(/no)</code>	
<code>/protocol:<protocol></code>	Overrides the default protocol to implement. You can choose <code>SOAP</code> , <code>HttpGet</code> , <code>HttpPost</code> , or a custom protocol, as specified in the configuration file.

Table 15.1 *continued*

Argument	Description
/username:<username> (/u)	Specifies the username to use when connecting to a server that requires authentication.
/password:<password> (/p)	Specifies the password to use when connecting to a server that requires authentication.
/domain:<domain> (/d)	Specifies the domain to use when connecting to a server that requires a domain name for authentication.

To create the proxy class for the Web service in Listing 15.14, the following command line is executed:

```
wsdl http://localhost/book/webservices/simple.asmx
```

Since only the URL for the Web service was specified, the wsdl utility generated the default files. By default, the proxy class file is named after the namespace. In this case, the generated file is named `SendMail.cs`. It contains C# code to generate a SOAP request to the Web service. If no language is specified as an argument to `wsdl.exe`, C# code is generated by default.

The entire generated proxy class is shown in Listing 15.14. Notice that the `SendMail` class implements the `System.Web.Services.Protocols.SoapHttpClientProtocol`. This class provides the SOAP transport plumbing. Also, the function definitions mimic those public methods of the Web service.

Listing 15.14 The Generated Web Service Proxy Class

```
//-----
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.0.2615.1
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
//-----

//
// This source code was auto-generated by wsdl, Version=1.0.2615.1.
//
namespace remailer {
    using System.Xml.Serialization;
    using System;
    using System.Web.Services.Protocols;
    using System.Web.Services;
```

Listing 15.14 continued

```
[System.Web.Services.WebServiceBindingAttribute(Name="SendMailSoap",
    Namespace="http://tempuri.org/")]
public class SendMail :
    System.Web.Services.Protocols.SoapHttpClientProtocol {

    public SendMail() {
        this.Url = "http://localhost/book/webservices/simple.asmx";
    }

    [System.Web.Services.Protocols.SoapMethodAttribute(
        "http://tempuri.org/Send", MessageStyle=
        System.Web.Services.Protocols.SoapMessageStyle.ParametersInDocument)]
    public void Send(string from, string to,
                    string subject, string messageText) {
        this.Invoke("Send", new object[] {from,
                                         to,
                                         subject,
                                         messageText});
    }

    public System.IAsyncResult BeginSend(string from,
                                         string to, string subject, string messageText,
                                         System.AsyncCallback callback, object asyncState) {
        return this.BeginInvoke("Send", new object[] {from,
                                                       to,
                                                       subject,
                                                       messageText}, callback, asyncState);
    }

    public void EndSend(System.IAsyncResult asyncResult) {
        this.EndInvoke(asyncResult);
    }
}
```

To use the proxy class from the ASP.NET application, it must be compiled and placed in the application's /bin directory. After this is done, the namespace containing the proxy class is recognized from anywhere in the application, and you can code against it, just as you can code against any other class. The following command line can be used to compile the proxy class in Listing 15.14:

```
csc /target:library SendMail.cs
```

If you forget to specify the /target:library option, you get an error from the compiler: No entry point found in SendMail.

Using Precompiled Web Services

In addition to creating the code for a Web service inline on an .asmx page, you can easily link to a precompiled managed class by placing the following code in an .asmx file:

```
<%@ WebService Language="VB" class="Acme.Sendmail"%>
```

This code assumes that you have a precompiled `SendMail` class in the .NET application. The assembly must be placed in the /bin directory of your application in order for the Web service to function.

The main difference between writing code inline and using a precompiled class appears the first time the Web service is used. In the case of inline code, the service will be compiled just-in-time (JIT) and served the first time it is called after its last code change. The precompiled class will not need to be recompiled.

Consuming Web Services from a Web Form

You have now analyzed the WSDL contract, built a proxy class, compiled the proxy class, and placed the resulting library in the /bin directory of the ASP.NET application you want to use to invoke the Web service. The next step is to actually write the code to consume. This code is very simple because consuming a Web service is no different from calling any other local method. Listing 15.16 contains a sample application that consumes the remailer Web service.

Listing 15.15 Consuming the Remailer Web Service

```
<% @Page Language="C#" %>
<%@ Import Namespace="remailer" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >

        public void Submit_Click(Object sender, EventArgs E)
        {

            string sFrom = txtFrom.Text;
            string sTo = txtTo.Text;
            string sSubject = txtSubject.Text;
            string sBody = txtMessage.Text;

            SendMail service = new SendMail();
            service.Send(sFrom, sTo, sSubject, sBody);
        }
    </script>
</HEAD>
<BODY>
    <Form>
        <Table>
            <Tr>
                <Td>From:</Td>
                <Td><input type="text" name="txtFrom" /></Td>
            </Tr>
            <Tr>
                <Td>To:</Td>
                <Td><input type="text" name="txtTo" /></Td>
            </Tr>
            <Tr>
                <Td>Subject:</Td>
                <Td><input type="text" name="txtSubject" /></Td>
            </Tr>
            <Tr>
                <Td>Message:</Td>
                <Td><input type="text" name="txtMessage" /></Td>
            </Tr>
            <Tr>
                <Td colspan="2" style="text-align: right;">
                    <input type="button" value="Send" onclick="Submit_Click(this, event)" />
                </Td>
            </Tr>
        </Table>
    </Form>
</BODY>

```

Listing 15.15 continued

```
MyLabel.Text = "Your message has been sent.";  
}  
</script>  
  
</HEAD>  
<BODY>  
  
<h1>Consuming Web Services</h1>  
<hr>  
  
<form runat="server" id=form1 name=form1>  
    <table>  
        <tr>  
            <td colspan=2>  
                <h2><asp:Label id="MyLabel" runat="server">  
                    Please fill out all fields.  
                </asp:Label></h2>  
            </td>  
        </tr>  
        <tr>  
            <td>  
                Your Email:  
            </td>  
            <td>  
                <asp:TextBox id="txtFrom" runat="server"></asp:TextBox>  
            </td>  
        </tr>  
        <tr>  
            <td>  
                Send to:  
            </td>  
            <td>  
                <asp:TextBox id="txtTo" runat="server"></asp:TextBox>  
            </td>  
        </tr>  
        <tr>  
            <td>  
                Message Subject:  
            </td>  
            <td>  
                <asp:TextBox id="txtSubject" runat="server"></asp:TextBox>  
            </td>  
        </tr>  
        <tr>  
            <td>  
                Message:  
            </td>  
            <td>
```

Listing 15.15 continued

```
<asp:TextBox TextBoxMode="MultiLine" id="txtMessage" runat="server">
</asp:TextBox>
</td>
</tr>
<tr>
<td>
<input type="submit" OnServerClick="Submit_Click" runat="server">
</form>
<hr>

</BODY>
</HTML>
```

Consuming Web Services from a Windows Form

A complete discussion of consuming Web services from Windows forms is beyond the scope of this book. However, the process is quite similar to consuming Web services from a Web form. If you compile the proxy class into your application, you can access the Web service namespace the same way you would any other local class.

Note that the consumption of Web services is not limited to the .NET framework, or even to the Microsoft platform. A SOAP toolkit is available for Visual Studio 6.0 that enables standard Active Server Pages (ASP) application to use Web services. Because the .NET architecture is completely open, Web services can be consumed from any platform.

Generating a Discovery File

Having a series of Web services running from a .NET application can greatly enhance its interoperability. However, unless programmatic discovery of Web services is enabled, consuming applications (that is, client applications) will need to know the precise location of your Web services. This may be desirable if your Web services are private or if you otherwise do not want to publish them to the world. However, programmatic discovery of Web services is built into the .NET framework.

Programmatic discovery is achieved through the creation of a discovery file (that is, a file with the .disco extension). A .disco file is an eXtensible Markup Language (XML) document that describes where to find more information about the Web service. Listing 15.16, from a Microsoft white paper on Web service discovery, shows the basic format of a discovery document.

Listing 15.16 Discovery File Format

```
<disco:discovery>
<disco:contractRef> ref='folder/discovery' />
<disco:discoveryRef ref='folder/discovery' />
```

```
<-- elements from other namespaces -->
</disco:discovery>
```

A discovery document for the SendMail Web service from Listing 15.17 would look like Listing 15.17. The Web Service Description Language (WSDL) contract for your Web service is provided in the `contractRef` attribute. You can add as many Web services to the discovery document as needed by specifying more `contractRef` tags. `discoveryRef` is used for other types of discovery documents.

Listing 15.17 Discovery File for the SendMail Web Service

```
<?xml version="1.0"?>
<discovery xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
            xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
            xmlns="http://schemas.xmlsoap.org/disco/">
    <contractRef ref="http://localhost/ASP.NET/WebServices/Sample.asmx?wsdl"
                  docRef="http://localhost/ASP.NET/WebServices/Sample.asmx"
                  xmlns="http://schemas.xmlsoap.org/disco/scl/" />
</discovery>
```

Providing the uniform resource locator (URL) to the `.disco` file will enable remote developers to programmatically discover your Web services. However, rather than provide a potentially long and confusing URL, you can link the discovery document to any Hypertext Markup Language (HTML) document by using syntax very similar to that which you use to attach a cascading style sheet to an application. The link is placed inside the HTML `<head>` tag, as shown in Listing 15.18.

Listing 15.18 Linking a Discovery File to an HTML Document

```
<HEAD>
<link type='text/xml' rel='alternate' href='MyWebService.disco' />
</HEAD>
```

By using this method, it is possible to link your discovery document to the default document for your Web application or domain. Programmatic interrogation would then be performed directly at the root.

Dynamic Discovery

In the Microsoft .NET framework, it is possible to enable dynamic Web service discovery. Placing a dynamic discovery document in a directory enables dynamic discovery of that directory and subdirectories. This is much easier than creating and maintaining a discovery file for each of your Web services.

The syntax of a dynamic discovery document is very similar to that of a standard discovery document. A `<dynamicDiscovery>` tag is used to enable dynamic discovery, and `<exclude>` tags are used to remove any subdirectories you do not want to be searched for in Web services. A sample dynamic discovery document is shown in Listing 15.19.

Listing 15.19 A Sample Dynamic Discovery Document

```
<?xml version="1.0" ?>
<dynamicDiscovery xmlns="urn:schemas-dynamicdiscovery:disco.2000-03-17">
<exclude path="SecretServices" />
</dynamicDiscovery>
```

You deploy a dynamic discovery document the same way you do a standard discovery document. The exact URL for the discovery document must be provided, unless the document is linked to the default document of your site.

NOTE

Because Web services operate over standard HTTP, a Web service is secured in the same manner as any Web page. A complete discussion of securing Web applications can be found in Chapter 17, “Securing and Deploying an ASP.NET Application.”

CHAPTER 16

Configuring and Optimizing an ASP.NET Application

This chapter describes how to configure and optimize ASP.NET applications. We will start by looking at the ASP.NET configuration system and then we will look at how to optimize applications by using the ASP.NET caching services.

Configuring ASP.NET Applications

ASP.NET provides an extensible, human-readable XML (extensible Markup Language)-based hierarchical configuration system. You can easily make configuration changes and extend the configuration to add any additional functionality your applications may require. In this section, we will discuss the configuration system in ASP.NET including the `machine.config` and `web.config`, two of the configuration files used by the Microsoft .NET Framework. Additionally, we will look at how you can store application data in the `web.config`, and how to display a custom user-friendly error page to users.

The `machine.config` File

The main configuration system file that affects all applications installed on the server is named `machine.config`, and it is located in the `config` directory in the Microsoft .NET Framework's installation directory. If you used the default installation, `machine.config` is located in the following directory:

`C:\WINNT\Microsoft.NET\Framework\[version number]\CONFIG`

The web.config File

Besides the `machine.config` file, each application can optionally have its own configuration file, called `web.config`. In this file you can override settings that are set in `machine.config`. In most cases, `web.config` is the file that you will work with when developing applications because often you do not want configurations for one application to affect all systems on the server.

NOTE

You can mark settings in the `machine.config` file so that they cannot be overridden by application-specific `web.config` settings. You do this by using the `<location>` tag and the `allowOverride` attribute. This can be an excellent security feature when you are securing your server.

Storing Application Settings

You can use the `web.config` file to store application settings in a place consistent with the rest of your configuration settings. This is particularly useful for settings that are going to be used in multiple spots in the application. An example of this would be storing the connection string to your data store. By placing the connection string in the configuration system, it can be easily changed without requiring a recompilation of the application. The application data is stored in the `AppSettings` section of the `web.config` file. Listing 16.1 and 16.2 demonstrate how to store the connection string in the `web.config` file.

Listing 16.1 Storing Application Data in the Web.Config File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

    <appSettings>
        <add key="DSN" value="server=localhost;uid=sa;pwd=;database=pureasp" />
    </appSettings>

</configuration>
```

Listing 16.2 Accessing AppSettings Information from a Component (C#)

```
using System;
using System.Configuration;

namespace PureASP {

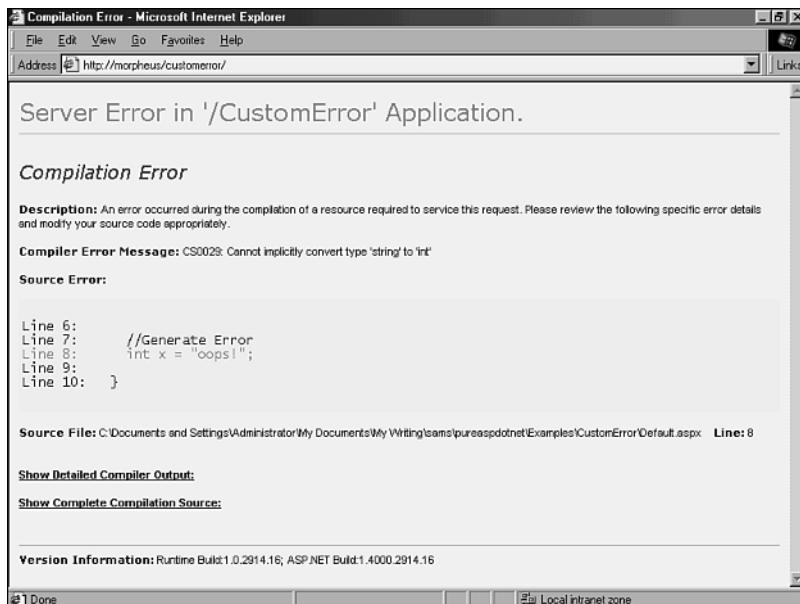
    public class AppSettings {
```

Listing 16.2 continued

```
public static string GetValueFromConfig(string Key) {  
    return (String) ConfigurationSettings.AppSettings[Key];  
}  
  
}  
}
```

Custom Error Reporting

When an application encounters an error, it is not very pleasant to display the raw error page, which is shown in Figure 16.1. First, it is not very user-friendly and it may confuse many users. Second, it could potentially be a security risk because the error page displays source code.

**Figure 16.1**

A standard error page.

The ASP.NET configuration system allows you to easily display a customer error page that is user-friendly and secure, like the one shown in Figure 16.2.

**Figure 16.2**

A custom error page.

You set the error page by adding a `customErrors` tag in the `System.Web` section of the `web.config` file. The `customErrors` tag has two attributes: `Mode` and `defaultRedirect`. The `Mode` attribute indicates whether the custom error page is enabled and whether it is available only to remote users. Table 16.1 summarizes the different values that `Mode` can be set to. The second attribute, `defaultRedirect`, indicates the filename of the custom error page. Listing 16.3 shows the `web.config` file for using an error page called `ErrorPage.htm`.

Table 16.1 Mode Attribute Values

Value	Description
On	Custom errors are enabled.
Off	Custom errors are disabled.
RemoteOnly	Custom errors are enabled but are displayed only to remote clients.

Listing 16.3 Assigning a Custom Error Page in the web.config File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <system.web>
    <customErrors mode="On" defaultRedirect="ErrorPage.htm" />
  </system.web>

</configuration>
```

Optimizing ASP.NET Applications

One of the benefits of ASP.NET is its great performance. However, you can improve the performance even more by using some of the features built into the framework. This section discusses how you can optimize the performance of ASP.NET applications by using the ASP.NET caching services.

The ASP.NET Caching Services

ASP.NET offers three types of caching: output caching, fragment caching, and data caching. The following sections discuss each of these in detail.

Output and Fragment Caching

Output and fragment caching allow you to store all or portions of an ASP.NET Web form in the cache. As long as the page remains cached, all requests for the page are returned to the cached Hypertext Markup Language (HTML) version of the page. This could potentially speed up your application enormously because it means that the ASP.NET processing of the page is completely bypassed at request time.

Output caching involves the caching of an entire Web form. When a user requests a page, such as default.aspx, the entire page's outputted HTML is placed in the cache. You add output caching to a page by adding the following directive at the top of the Web form:

```
<%@ OutputCache Duration="60" %>
```

This tag causes the page to be placed in the cache for 60 seconds. After the 60 seconds has expired, the next request is processed and replaced in the cache for an additional 60 seconds.

There are times when output caching doesn't make sense. When you have pages that change often, such as pages that have user personalization, it would make no sense to cache the output of the page. If you did, when John Doe visited the site, the page would be displayed personalized for him. At the same time, the page would be placed in the cache and all requests that were made while the page remained in the cache would be served the page personalized for John Doe. For pages like this, it would be better to use fragment caching and cache the portions of the page that are not personalized.

Fragment caching allows you to cache portions of a page. You break the page up through by using user controls (described in Chapter 10, "Encapsulating ASP.NET Page Functionality with User Controls"). The procedure to add fragment caching is very similar to the procedure for implementing output caching, except instead of placing the directive at the top of the Web form, you place it at the top of the user control.

There are times when storing one version of a user control is not sufficient. For instance, if you have a menu control that highlights the item that is selected, storing one version would not make sense. However, if you could store a version for each item in the list, you could still cache the results.

The ASP.NET configuration system supports storing multiple versions of a page or user control in a page by setting the `VaryByParam` attribute in the `OutputCache` directive. The following example stores a different version of the menu user control for each `CategoryId` passed to the control:

```
<%@ OutputCache Duration="3600" VaryByParam="CategoryId" %>
```

Data Caching

Data caching allows you to store data in the cache by using the `Cache` object so that you can limit the number of calls you have to make to assemblies or data sources. Data caching is a good solution for caching application data programmatically. A good example of using data caching would be to store data returned from a database. This way, you can cache the results from the data query and eliminate the need to re-query the database each time the page is loaded. You can place an item in the cache, using the following C# syntax:

```
Cache[ "Products" ] = GetProducts();
```

or the following Visual Basic.NETsyntax:

```
Cache( "Products" ) = GetProducts()
```

CHAPTER 17

Securing and Deploying an ASP.NET Application

An ASP.NET application can consist of several distinct sections. There is a series of .aspx pages, which provide the graphical user interface. In an *N*-tier application, there are also likely to be various components that encapsulate business logic and provide access to a database. Finally, configuration files, such as config.web, provide application-specific settings.

Before the creation of the Microsoft .NET framework, it was difficult to deploy an application on a standard Component Object Model (COM)/Active Server Pages (ASP)-based platform. To ensure successful deployment, all the developers on a project needed to coordinate with one another. If the application used dynamic link libraries (DLLs), and a change to the DLL required breaking binary compatibility, all DLLs referencing the broken DLL would need to be recompiled so that they would reference the new type library. Additionally, if the application was running when the compatibility was broken, the DLL would be locked. To deploy the new DLL, the Internet Information Server (IIS) application needed to be stopped for the duration of the deployment, potentially bringing down the site unless component load balancing was being used across more than one server. Developers labeled this process not so affectionately “DLL hell.”

Microsoft has solved virtually all deployment problems that plagued its previous development platforms. Deploying an ASP.NET application is as simple as copying a directory of files. This chapter details the features in the Microsoft .NET framework that make deploying and configuring an ASP.NET application easier than in previous versions of ASP.

ASP.NET Installation Benefits

The Microsoft .NET framework has a few exceptional characteristics that make deploying an ASP.NET application easy and painless.

No files are locked in a running ASP.NET application. Even DLLs, which would most certainly be locked in previous versions of ASP, are copied into and run from memory. Additionally, DLLs do not need to be registered with the operating system. Also, configuration of an ASP.NET application is performed through the use of human-readable XML-based configuration files.

Side-by-Side DLL Execution

The Microsoft .NET framework allows several different versions of a DLL to be installed on the same machine simultaneously. An application automatically loads the correct version of the DLL by first searching the application's /bin subdirectory and then searching the global assembly cache. This alone eliminates most of the versioning problems associated with managing DLLs.

Deploying ASP.NET Pages

Creating the Virtual Directory

The first step in correctly deploying an ASP.NET application is to create the virtual directory in the Internet Services Manager (ISM). In Windows 2000, this is a straightforward process. In a default configuration, the ISM can be found under Administration Tools in the Control Panel or in the Windows 2000 Start Menu. When the ISM is open, follow these steps to create the site's virtual directory:

1. With the ISM open, right-click on Default Web Site and select New and then Virtual Directory. The Virtual Directory Creation Wizard opens; it walks you through the steps of creating a virtual directory.
2. First, the wizard asks you to provide an alias—that is, the name of the virtual directory. When someone is accessing your site, the alias will be in the following location: `http://localhost/Alias`.
3. The wizard then asks for the hard disk physical directory where the files for your application are located. If the application files are not present on the machine yet, choose the directory where they will be located after they are installed. The directory must be present before you run the wizard. The wizard offers no means to create the directory.
4. Next, the wizard asks you to specify access permissions to the site. Choose the default values for now; you can modify them later, if needed. The virtual directory is then created. If the application files are already installed, then the application can be accessed via `http://site name/virtual directory alias`. (For instance, if the new virtual directory named test was created on `http://www.eraserver.com`, the final site location is `http://www.eraserver.com/test`.)

Deploying ASP.NET Applications by Using Standard Internet Protocols

When the virtual directory is in place, you can deploy an ASP.NET application in a number of ways. One of the easiest ways is to use a tool such as Microsoft Visual InterDev, which uses FrontPage extensions to IIS to copy the application files from the development machine to the application directory on the server.

Another method for deploying an ASP.NET application is to create a File Transfer Protocol (FTP) directory in the ISM that points to the directory where the application files are located. Then, to perform updates to the application, you simply use FTP to copy the new files over the old ones.

Scripting Deployment

Because application deployment no longer requires direct access to the Web server with the introduction of the Microsoft .NET framework, the entire deployment can be scripted. A standard batch program, shell script, or any other scripting method can be used to script the copying of the necessary files.

Deploying Components

Because DLLs are no longer locked when in use and since the .NET framework does not require assemblies to be registered with the operating system, deploying a component is as simple as deploying a Web page. You only need to copy the compiled DLL to the /bin subdirectory of the ASP.NET application. If there is no /bin subdirectory currently present, you should create one.

Setting Up Staged Deployment

The text so far in this chapter assumes that you do not have direct access to the Web server file system. This might be the case if you are working with an Internet service provider and sharing a server with the provider's other clients.

However, if the Web servers exist on your network or you otherwise have direct access to them, there are much better deployment models. Many companies choose to implement a three-tiered model of application deployment, in which the tiers are development, testing, and production.

For the most part, the development tier consists of a machine or a set of machines configured as closely to the production environment as possible. Developers are allowed full access to almost all aspects of the machine. Applications are created and go through the first wave of testing in this environment.

The servers in the testing tier must be configured exactly as the production Web servers are configured. When an application in development is ready to be tested, it is placed in the test environment. This enables the developer and anyone else who is testing the application to study the performance in an environment that is identical to the production environment. When the necessary bug fixes are performed and the application is

deemed ready for production, everyone can be confident that the application will perform as well in the production environment as it has in the testing environment.

There are a few important items to note about the testing tier. First, developers should have as little access to the environment as possible. Permissions should be set so that each developer can perform modifications only on his or her own application. Developers should not be allowed to modify global IIS settings or other elements of the operating system. This helps to ensure that the environment remains as sterile as possible. Second, the testing tier should mirror production in every manner possible. If the production tier consists of several servers in a cluster, then the testing tier should be configured the same way.

Ideally, developers should have no access whatsoever to the production tier. A migration tool (such as the Microsoft Site Server Publishing Wizard) should be used to copy the files from the test environment to production. It is imperative that the production environment remain sterile, and all application changes must be logged extensively to ensure the fewest possible problems.

Three-Tier Hardware Requirements

At first, the cost of a three-tier deployment model might seem prohibitive. However, the development tier can be the developer's workstations. A production server (or servers) will already be present. Thus, the only additional costs come from the test tier. But even if load balancing is required in order to mirror the production environment, only two additional are required.

Benefits of Staged Deployment

There are several benefits to staged deployment. First, by having a clean test environment configured identically to the production environment, you can be confident that a tested application will work flawlessly in the production environment. Second, because the only person who needs access to the production environment is the network administrator, the production machines can be secured well.

Securing an Application

Security in ASP.NET applications is very similar to security in previous versions of ASP. Almost all the security is provided on the operating system, file system, and network levels. In fact, there are no known holes in security in the .NET framework. However, misconfiguration of the production environment from the network to the operating system can open an environment to a number of attacks, from denial of service to old-fashioned Trojan horse attacks.

Security by Design

Designing your application and environment with security in mind from the very beginning is the easiest way to ensure that your environment is as secure as possible. A complete discussion of security is not possible in the scope of this book. However, the following sections list some general guidelines for network and operating system security.

Network Security

The outside world should have little access to the production environment as is necessary for your applications to run. At the very least, a firewall should be placed between your production machines and the Internet. Firewalls are designed to filter incoming and outgoing network traffic. For a typical ASP.NET application, port 80 (that is, the HTTP port) and perhaps port 443 (that is, the secure HTTP port) should be exposed to the outside world through the firewall.

Quite often, production Web servers have several network interface cards (NICs) installed. One NIC should be used for Internet connections. This interface should be limited by Internet Protocol (IP) filtering to accept only the ports that are allowed through the firewall. The other NIC should be connected to the internal company network.

Operating System Security

It is important for the administrator of the production environment to monitor all security alerts and install security-related hot fixes and service packs as they become available. Also, modifications to user and file system objects should be audited (that is, logged) in order to detect attacks. A number of security packages are available that monitor the operating system for common attacks and alert the system administrator immediately. The CERT Coordination Center (www.cert.org) is an excellent resource to help you remain up to date on the latest security threats because it is one of the major reporting centers for security problems. Additionally, Microsoft has a Security Best Practices page, located at www.microsoft.com/technet/security/bestprac.asp, where you can find information on just about all aspects of security and how it relates to Microsoft products.

Windows Authentication

Securing individual Web pages or directories can be achieved in several different ways. If a Web resource needs to be secure, the first step is to remove anonymous access for the resource. Anonymous access is provided by default, and it is present unless it is turned off.

The config.web file can be used to secure any ASP.NET files (that is, any files registered to be handled by `xspisapi.dll`). However, image files, HTML files, and many other files remain accessible to all users, even if access is denied in config.web. For this reason, security should be handled by the standard IIS Internet Security Manager first for all files. Then, additional security settings should be provided in config.web solely for ASP.NET resources.

Basic Authentication

The simplest method of authentication is basic authentication. Unless a user has already authenticated, the user is presented with a dialog box asking them to present username/password credentials for the specified resource. Basic authentication works across almost all browsers, but the password is transmitted over the network as base-64-encoded text and can be intercepted at any point between the user's computer and

the Web server. For this reason, basic authentication is used only where security is not a priority.

NTLM Authentication

Windows NT LAN Manager (NTLM) authentication works much like basic authentication except that the password is never actually transmitted over the network in clear-text format. Instead, a challenge-response password authentication system is used. In a challenge-response authentication system, the server sends the client a challenge in the form of a random set of characters. Then the client computes a response, which is a function of both the user's password and the challenge sent from the server. By decoding the client response, the server is able to verify the user's password and grant or deny access.

Form-Based Authentication

Quite often an application requires a user to log into the site in order to identify himself or herself. Generally, security isn't as much an issue as identifying the user in order to personalize content and track resource usage. This is called form-based authentication, and in this model, user information such as the user name is stored in a cookie on the user's machine.

In previous versions of ASP, every page checked to make sure that the cookie was present and that the user had been authenticated. ASP.NET provides form-based authentication to manage this process in config.web and through the use of a login page.

Configuring Authentication with config.web

As shown in Chapter 16, "Configuration and Optimization of an ASP.NET Application," config.web is at the heart of configuration changes in an ASP.NET application. Therefore, it is not surprising that form-based authentication is configured in the same place. Both the login page URL and all the pages in the site that require authentication are configured in config.web.

Specifying the Login URL

To use form-based authentication, the first step is to specify the cookie authentication mode in config.web. This is shown in Listing 17.1.

Listing 17.1 Specifying the Login URL

```
<security>
  <authentication mode="Cookie">
    <cookie cookie="AppAuth" loginurl="Login.aspx"
           decryptionkey="autogenerate">
      </cookie>
    </authentication>
  </security>
```

Listing 17.1 sets the authentication mode to use cookies and then specifies the login URL, which is the page that will provide the actual authentication logic.

Indicating Secure Pages

For each page that requires form-based authentication, you create an entry in config.web that looks like the entry in Listing 17.2.

Listing 17.2 Marking Secure Pages for Authentication

```
<location path="MyPage.aspx">
  <security>
    <authorization>
      <deny users="?" />
    </authorization>
  </security>
</location>
```

Listing 17.2 specifies MyPage.aspx as secure. If an unauthenticated user attempts to access the page, he or she will be redirected to the Login.aspx page. When the user logs in, he or she will be redirected back to the page that he or she originally attempted to access.

By using the deny tag, it is possible to restrict specific users from accessing a page.

Building a Login Page

The login page performs whatever operations are necessary to authenticate the user. The operations performed are completely up to your discretion and obviously depend on individual application requirements. When a user has been successfully authenticated, the following method is called:

```
CookieAuthentication.RedirectFromLoginPage([CustomerIdentifier],
                                           [Remember Login])
```

The first argument to the RedirectFromLoginPage function (that is, CustomerIdentifier) is an object used to identify the user. The second argument (that is, Remember Login) specifies whether to remember the user's authentication information in the future so that the user is not prompted to log in again.

PART III

MICROSOFT .NET REFERENCE BY NAMESPACE

- 18 System.Collections Reference
- 19 System.Data.SqlClient Reference
- 20 System.Web Reference
- 21 System.Web.UI.WebControls Reference



CHAPTER 18

System.Collections Reference

The `System.Collections` namespace contains classes that provide a common functionality for working with collections. This chapter provides a reference to the classes contained in the `System.Collections` namespace as well as the properties and methods of these classes.

The following block of code contains code common to all C# listings in this chapter; the code for the examples can be inserted below the comment in the `Page_Load()` event:

```
<% @Page Language="C#" %>

<HTML>
<HEAD>
    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {
            //Individual Example Code Is Placed Here

        }
    </script>
</HEAD>
<BODY>

<form runat="server">

    <asp:datagrid id=MyList runat="server">
    </asp:datagrid>

    <br><br>

    <asp:datagrid id=MyList2 runat="server">
```

```
</asp:datagrid>

<asp:label id=MyLabel runat="server"></asp:label><br>
<asp:label id=MyLabel2 runat="server"></asp:label>

</form>
</BODY>
</HTML>
```

Likewise, all Visual Basic.NET code examples can be placed into the following block of code in order to achieve functionality:

```
<% @Page Language="VB" %>

<HTML>
<HEAD>
    <script language="VB" runat="server" >
        Sub Page_Load(Source as Object, E as EventArgs)
            'Individual Example Code Is Placed Here

        End Sub
    </script>
</HEAD>
<BODY>

<form runat="server">

    <asp:datagrid id=MyList runat="server">
    </asp:datagrid>

    <br><br>

    <asp:datagrid id=MyList2 runat="server">
    </asp:datagrid>

    <asp:label id=MyLabel runat="server"></asp:label><br>
    <asp:label id=MyLabel2 runat="server"></asp:label>

</form>
</BODY>
</HTML>
```

The ArrayList Class

The `ArrayList` class provides a number of properties and methods that are used to work with an arraylist. Arraylists are zero-based in ASP.NET. The properties and methods of the `ArrayList` class are listed in Table 18.1.

Table 18.1 Properties and Methods of the ArrayList Class

Item	Description
Properties	
Capacity	Contains the allocated length of the arraylist
Count	Contains the number of items currently in the arraylist
IsFixedSize	Contains True if the arraylist is of a fixed size
IsReadOnly	Contains True if the arraylist is read-only and False if it is not read-only
IsSynchronized	Contains True if the arraylist is synchronized
Item[]	Stores an index of all the items in the arraylist
Methods	
Adapter()	Accepts a list and returns an arraylist
Add()	Adds an item to the arraylist and returns the newly added index
AddRange()	Adds a collection to the arraylist and returns the index of the first item added
BinarySearch()	Performs a binary search on the arraylist
Clear()	Clears all items from the arraylist
Clone()	Creates a duplicate of the arraylist
Contains()	returns true if the given object is in the current arraylist
CopyTo()	Copies all or part of the current arraylist to another arraylist that is passed in as an argument
FixedSize()	Returns an arraylist of a fixed size
GetRange()	Returns the range of the current arraylist
IndexOf()	Returns the index of an item in the arraylist
Insert()	Inserts an item into the arraylist
InsertRange()	Inserts a collection of items into the arraylist
LastIndexOf()	Returns the last index of an item in the arraylist
ReadOnly()	Returns a read-only copy of the current arraylist
Remove()	Removes an item from the arraylist
RemoveAt()	Removes an item from the arraylist by index
RemoveRange()	Removes a range of items from the arraylist
Repeat()	Creates an arraylist of specified size, with all elements containing the same object
Reverse()	Reverses items in the arraylist
SetRange()	Sets the range of the arraylist
Sort()	Sorts the items in the arraylist
Synchronized()	Synchronizes the arraylist
ToArray()	Converts the arraylist to an array of the type passed in as an argument; returns the new array
TrimToSize()	Changes the size of the arraylist to match the number of items currently in the arraylist

ArrayList.Capacity

Syntax

Int32 Capacity

Description

The Capacity property allows the size of the arraylist to be retrieved and set. If you attempt to set the size of the arraylist so that it is smaller than the current size of the arraylist, an exception is thrown.

Example

```
<html><head>
<script language="VB" runat="server">
    Sub Page_Load(Source As Object, E As EventArgs)
        If Not IsPostBack Then
            End If
        End Sub
    </script>
<form runat="server">
    <asp:Label id="MyLabel" runat="server" />
</form>
</body><html>
```

Listings 18.1 and 18.2 show how to both get and set the capacity of an arraylist.

Listing 18.1 Reading and Setting ArrayList Capacity (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");
items.Add("item3");

Items.Capacity = 5; //The default size is 16

MyLabel.Text = "Capacity is " + items.Capacity.ToString();
```

Listing 18.2 Reading and Setting ArrayList Capacity (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")
items.Add("item3")

items.Capacity = 5

MyLabel.Text = "Capacity is " + items.Capacity.ToString()
```

ArrayList.Count

Syntax

Int32.Count

Description

The Count property contains the number of items in the arraylist.

Example

Listings 18.3 and 18.4 show how to access the Count property of an arraylist.

Listing 18.3 Accessing the Count Property (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");
items.Add("item3");

MyLabel.Text = "Count is " + items.Count.ToString();
```

Listing 18.4 Accessing the Count Property (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")
items.Add("item3")

MyLabel.Text = "Count is " + items.Count.ToString()
```

ArrayList.IsFixedSize

Syntax

Boolean IsFixedSize

Description

The IsFixedSize property contains True if the arraylist is of a fixed size. This property is set to False by default.

Example

Listings 18.5 and 18.6 demonstrate how to access the ArrayList.IsFixedSize property.

Listing 18.5 Accessing the IsFixedSize Property (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");
```

Listing 18.5 continued

```
items.Add("item3");

MyLabel.Text = "IsFixedSize set to " + items.IsFixedSize;
```

Listing 18.6 Accessing the IsFixedSize Property (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")
items.Add("item3")

MyLabel.Text = "IsFixedSize is " + items.IsFixedSize.ToString()
```

ArrayList.IsReadOnly

Syntax

```
Boolean IsReadOnly
```

Description

The `IsReadOnly` property contains `True` if the `ArrayList` is read-only. This property is set to `False` by default. To change the read-only state of an `ArrayList`, you use the `ArrayList.ReadOnly()` method.

Example

Listings 18.7 and 18.8 demonstrate how to access the `ArrayList.IsReadOnly` property.

Listing 18.7 Accessing the IsReadOnly Property (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");
items.Add("item3");

MyLabel.Text = "IsReadOnly set to " + items.IsReadOnly;
```

Listing 18.8 Accessing the IsReadOnly Property (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")
items.Add("item3")

MyLabel.Text = "IsReadOnly set to " + items.IsReadOnly.ToString()
```

ArrayList.IsSynchronized

Syntax

```
Boolean IsSynchronized
```

Description

The `IsSynchronized` property contains `True` if the arraylist is synchronized and `False` otherwise.

Example

Listings 18.9 and 18.10 demonstrate how to access the `ArrayList.IsSynchronized` property.

Listing 18.9 Accessing the IsSynchronized Property (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");
items.Add("item3");

MyLabel.Text = "IsSynchronized set to " + items.IsSynchronized;
```

Listing 18.10 Accessing the IsSynchronized Property (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")
items.Add("item3")

MyLabel.Text = "IsSynchronized set to " + items.IsSynchronized.ToString()
```

ArrayList.Item

Syntax

```
Object Item ( Int32 index )
```

Description

The `Item` property enables a developer to access elements of the arraylist by using a numerical index. Attempts to access an element outside the current range of the arraylist cause an exception to be thrown.

Example

Listings 18.11 and 18.12 demonstrate how to use the `ArrayList.Item` property to both get and set values in the arraylist.

Listing 18.11 Accessing and Setting the Item[] Property (C#)

```
ArrayList items = new ArrayList();
items.Item[0] = "newitem";
MyLabel.Text = "First Item in ArrayList: " + (string)items[1];
```

Listing 18.12 Accessing and Setting the Item[] Property (Visual Basic.NET)

```
dim items = new ArrayList()
items.Item(0) = "newitem"
MyLabel.Text = "Second Item in ArrayList: " + items.Item(1)
```

ArrayList.Adapter()

Syntax

```
ArrayList Adapter( IList list )
```

Description

The Adapter() method is used to convert any list into an the Item[] Property . It accepts an IList object as an argument and returns an arraylist.

Example

Listings 18.13 and 18.14 demonstrate how to use ArrayList.Adapter(). After the last line executes, NewList contains all the entries in the original item's arraylist.

Listing 18.13 Converting a List with ArrayList.Adapter() (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");
items.Add("item3");

ArrayList NewList = new ArrayList();

NewList = ArrayList.Adapter(items);
```

Listing 18.14 Converting a List with ArrayList.Adapter() (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")
items.Add("item3")

dim NewList as new ArrayList()

NewList = ArrayList.Adapter(items)
```

ArrayList.Add()

Syntax

```
Int32 Add ( Object value )
```

Description

The `ArrayList.Add()` method is used to add an element to an arraylist. It automatically increases the size of the arraylist if needed, as long as the arraylist is not a fixed size.

Example

Listings 18.15 and 18.16 declare a new arraylist and then adds a few values.

Listing 18.15 Adding Objects to an ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");
```

Listing 18.16 Adding Objects to an ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")
```

ArrayList.AddRange()

Syntax

```
Void AddRange( ICollection c )
```

Description

`ArrayList.AddRange()` accepts a collection and adds the items in the collection to the arraylist. Like the `ArrayList.Add()` method, `ArrayList.AddRange` automatically expands the capacity of the arraylist unless the arraylist is of fixed size.

Example

Listings 18.17 and 18.18 create and fill two arrays, and then adds the second arraylist to the first. When the last line is executed, the items in the second arraylist are appended to the items in the first arraylist.

Listing 18.17 Expanding ArrayLists with AddRange() (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item1");

ArrayList NewList = new ArrayList();
```

Listing 18.17 continued

```
NewList.Add("newValue1");
NewList.Add("newValue2");

items.AddRange(NewList);
```

Listing 18.18 Expanding ArrayLists with AddRange() (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

Dim NewList as new ArrayList()
NewList.Add("newValue1")
NewList.Add("newValue2")

items.AddRange(NewList)
```

ArrayList.BinarySearch()

Syntax

```
Int32 BinarySearch( Object value, IComparer comparer )
Int32 BinarySearch( Object value )
Int32 BinarySearch( Int32 index, Int32 count, Object value, IComparer
➥comparer )
```

Description

The `BinarySearch()` method performs a binary search on the given arraylist.

The simplest method for performing a binary search on an arraylist is to pass in a search value only. This will search the entire arraylist for the value. Passing in a comparer as the second argument enables the developer to search more specifically. Finally, for large arraylists, passing in a starting index and range as the first two arguments will greatly reduce search time.

Example

Listings 18.19 and 18.20 demonstrate how to use the `ArrayList.BinarySearch()` method to find an arraylist element and pass the index to a label Web control.

Listing 18.19 Using ArrayList.BinarySearch to Find a Value (C#)

```
MyLabel.Text = items.BinarySearch("item2").ToString();
```

Listing 18.20 Using ArrayList.BinarySearch to Find a Value (Visual Basic.NET)

```
MyLabel.Text = items.BinarySearch("item2").ToString()
```

ArrayList.Clear()

Syntax

```
Void Clear()
```

Description

The ArrayList.Clear() method removes all items from the arraylist collection.

Example

Listings 18.21 and 18.22 demonstrate how to remove all items from an arraylist, by using the Clear() method.

Listing 18.21 Removing All Items from an ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
Items.Clear();
```

Listing 18.22 Removing All Items from an ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Clear()
```

ArrayList.Clone()

Syntax

```
Object Clone()
```

Description

The ArrayList.Clone() method creates a copy of the given arraylist. It returns a generic Object type of arraylist.

Example

Listings 18.23 and 18.24 demonstrate how to clone an existing arraylist and push it into another arraylist. Because ArrayList.Clone() returns a generic object, this method could be used to convert an arraylist to a different type of collection.

Listing 18.23 Making a Copy of an ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");

ArrayList NewList = (ArrayList) items.Clone();
```

Listing 18.24 Making a Copy of an ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()  
items.Add("item1")  
  
dim NewList as ArrayList  
  
NewList = items.Clone()
```

ArrayList.Contains

Syntax

Boolean Contains(Object item)

Description

The `ArrayList.Contains()` method accepts a search object as an argument and returns True if the object is found in the arraylist.

Example

Listings 18.25 and 18.26 demonstrate how to use `ArrayList.Contains()` to see if an arraylist contains a string value. In this example, the Boolean return is cast to a string and pushed into a label Web control.

Listing 18.25 Checking Whether an ArrayList Contains an Object (C#)

```
ArrayList items = new ArrayList();  
items.Add("item1");  
MyLabel.Text = "Does list contain this object: "  
    + items.Contains("item1").ToString();
```

Listing 18.26 Checking Whether an ArrayList Contains an Object (Visual Basic.NET)

```
dim items = new ArrayList()  
items.Add("item1")  
MyLabel.Text = "Does list contain this object: "  
    + items.Contains("item1").ToString()
```

ArrayList.CopyTo()

Syntax

Void CopyTo(Array array, Int32 arrayIndex)
Void CopyTo(Array array)
Void CopyTo(Int32 index, Array array, Int32 arrayIndex, Int32 count)

Description

The `ArrayList.CopyTo()` method is used to copy the contents of an arraylist to an array. The simplest way to do this is to pass the array as the only argument. This copies

the entire arraylist to the array. The array must be large enough to hold all the arraylist elements, or an exception will be thrown.

Passing in an index as the second argument copies all items from that index to the last item in the arraylist, inclusive. Finally, it is possible to pass in a starting index in the arraylist and also an index on the array, as well as a count. This copies a range of values from the arraylist, starting at the index and continuing to anyplace in the array.

Example

Listings 18.27 and 18.28 demonstrate how to use the `ArrayList.CopyTo()` method to copy an arraylist to a string array.

Listing 18.27 Copying an Arraylist to an Array (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

String[] MyArray = new String[2];
items.CopyTo(MyArray);
MyLabel.Text = "First item of ArrayList is " + MyArray[0];
```

Listing 18.28 Copying an Arraylist to an Array (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

dim MyArray(4) as String

items.CopyTo(MyArray)

MyLabel.Text = "Third item of ArrayList is " + MyArray(2)
```

ArrayList.FixedSize()

Syntax

```
IList FixedSize( IList list )
ArrayList FixedSize( ArrayList list )
```

Description

The `ArrayList.FixedSize()` method accepts either a list or an arraylist as an argument and returns a fixed-size version of the list or arraylist.

Example

Listings 18.29 and 18.30 demonstrate how to create a simple arraylist and then create a fixed-size arraylist from the original.

Listing 18.29 Creating a Fixed-Size ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

ArrayList NewList = new ArrayList();
NewList = ArrayList.FixedSize(items);
//NewList.Add("test"); //this line, if uncommented, would throw exception
```

Listing 18.30 Creating a Fixed-Size ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")
dim NewList as new ArrayList()

NewList = ArrayList.FixedSize(items)
```

ArrayList.GetRange()

Syntax

```
ArrayList GetRange( Int32 index, Int32 count )
```

Description

The `ArrayList.GetRange()` method accepts an index and a range (as count). It returns an arraylist of the original arraylist, starting at the given index and range.

Example

Listings 18.31 and 18.32 demonstrate how to use the `ArrayList.GetRange()` method to create a new arraylist that is a subset of the original.

Listing 18.31 Creating a Subset of the Original ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

ArrayList NewList = new ArrayList();
NewList = items.GetRange(1, 2);
```

Listing 18.32 Creating a Subset of the Original ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

dim NewList as new ArrayList()
NewList = items.GetRange(1, 2)
```

ArrayList.IndexOf()

Syntax

```
Int32 IndexOf( Object value )
Int32 IndexOf( Object value, Int32 startIndex )
Int32 IndexOf( Object value, Int32 startIndex, Int32 endIndex )
```

Description

The `ArrayList.IndexOf()` method is used to find an index of the given value in the arraylist.

There are two overloaded `IndexOf()` methods. The first enables the developer to enter a starting index, and the second enables the developer to specify both a starting and ending index. In large arraylists, passing in these extra arguments could greatly reduce search time.

If the value is not found in the arraylist, `ArrayList.IndexOf()` returns -1. If the value is in the arraylist more than once, `ArrayList.IndexOf()` returns the index of the first instance.

Example

Listings 18.33 and 18.34 demonstrate how to use `ArrayList.IndexOf()` to return the index of an element in the arraylist. The return value is pushed into a label Web control.

Listing 18.33 Returning the Index of a Value (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

MyLabel.Text = "IndexOf 'item2' is: " + items.IndexOf("item2");
```

Listing 18.34 Returning the Index of a Value (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

MyLabel.Text = "IndexOf 'item2' is: " + items.IndexOf("item2").ToString()
```

ArrayList.Insert()

Syntax

```
Void Insert( Int32 index, Object value )
```

Description

The `ArrayList.Insert()` method is used to insert a value into an arraylist at a given index. The size of the original arraylist will be increased and reindexed automatically.

Example

Listings 18.35 and 18.36 demonstrate how to use `ArrayList.Insert()` to insert a value into an existing arraylist.

Listing 18.35 Inserting a Value into an Arraylist (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

items.Insert(1, "InsertedValue");
```

Listing 18.36 Inserting a Value into an Arraylist (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

items.Insert(1, "InsertedValue")
```

ArrayList.InsertRange()

Syntax

```
Void InsertRange( Int32 index, ICollection c )
```

Description

The `ArrayList.InsertRange()` method inserts a collection of items into an arraylist at the given index.

Example

Listings 18.37 and 18.38 insert an arraylist of values into another arraylist.

Listing 18.37 Inserting a Collection of Values into an Arraylist (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

ArrayList NewList = new ArrayList();
NewList.Add("newValue1");
NewList.Add("newValue2");

items.InsertRange(1, NewList);
```

Listing 18.38 Inserting a Collection of Values into an Arraylist (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
```

Listing 18.38 continued

```
items.Add("item2")

dim NewList as new ArrayList()
NewList.Add("newValue1")
NewList.Add("newValue2")

items.InsertRange(1, NewList)
```

ArrayList.LastIndexOf()**Syntax**

```
Int32 LastIndexOf( Object value )
Int32 LastIndexOf( Object value, Int32 startIndex )
Int32 LastIndexOf( Object value, Int32 startIndex, Int32 endIndex)
```

Description

The `ArrayList.LastIndexOf()` method returns the last index of the value in the arraylist. Overloaded methods of `LastIndexOf()` enable the developer to pass in a start index and an end index to limit the range and increase search performance.

Example

Listings 18.39 and 18.40 demonstrate how to get the index of the last instance of a value in an arraylist. The value is pushed into a label Web control.

Listing 18.39 Getting the Last Index of a Value in an Arraylist (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");
items.Add("item3");

MyLabel.Text = "Last Index of 'item1' is: " + items.LastIndexOf("item1");
```

Listing 18.40 Getting the Last Index of a Value in an Arraylist (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")
items.Add("item3")

MyLabel.Text = "Last Index of 'item2' is: "
               + items.LastIndexOf("item2").ToString()
```

ArrayList.ReadOnly()

Syntax

```
IList ReadOnly( IList list )
ArrayList ReadOnly( ArrayList list )
```

Description

The `ArrayList.ReadOnly()` method accepts either an `IList` object or an `ArrayList` as an argument and returns a read-only copy of the same type.

Example

Listings 18.41 and 18.42 demonstrate how to retrieve a read-only copy of an `ArrayList`.

Listing 18.41 Getting a Read-Only Copy of an ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

ArrayList NewList = ArrayList.ReadOnly(items);
//NewList.Add("bla"); //This line throws an error if not commented
```

Listing 18.42 Getting a Read-Only Copy of an ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

dim NewList as new ArrayList()

NewList = ArrayList.ReadOnly(items)
```

ArrayList.Remove()

Syntax

```
Void Remove( Object obj )
```

Description

The `ArrayList.Remove()` method removes the first instance of an object found in an `ArrayList`. An exception is thrown if the object is not found in the `ArrayList`.

Example

Listings 18.43 and 18.44 demonstrate how to use the `ArrayList.Remove()` method to remove an object from an `ArrayList`.

Listing 18.43 Removing an Item from an ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

items.Remove("item1");
```

Listing 18.44 Removing an Item from an ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

items.Remove("item2")
```

ArrayList.RemoveAt()

Syntax

```
Void RemoveAt( Int32 index )
```

Description

The `ArrayList.RemoveAt()` method removes an item from an arraylist by index passed in as an argument.

Example

Listings 18.45 and 18.46 demonstrate how to remove the first item from an arraylist.

Listing 18.45 Removing the First Item from an ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

items.RemoveAt(0);
```

Listing 18.46 Removing the First Item from an ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

items.RemoveAt(1)
```

ArrayList.RemoveRange()

Syntax

```
Void RemoveRange( Int32 index, Int32 count )
```

Description

The `ArrayList.RemoveRange()` method removes a range of items from an arraylist.

Example

Listings 18.47 and 18.48 demonstrate how to remove the first two items of an arraylist.

Listing 18.47 Removing a Range of Items from an ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

items.RemoveRange(0, 2);
```

Listing 18.48 Removing a Range of Items from an ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

items.RemoveRange(0, 2)
```

ArrayList.Repeat()

Syntax

```
ArrayList Repeat( Object value, Int32 count )
```

Description

The `ArrayList.Repeat()` method returns an arraylist of size Count (second argument), which contains the value passed in as the first argument.

Example

Listings 18.49 and 18.50 demonstrate how to use the `ArrayList.Repeat()` method to return an arraylist with 10 items, all containing the value test.

Listing 18.49 Creating a New ArrayList with Default Items, Using Repeat() (C#)

```
ArrayList NewList = new ArrayList();
NewList = ArrayList.Repeat("test", 10);
```

Listing 18.50 Creating a New ArrayList with Default Items, Using Repeat() (Visual Basic.NET)

```
dim NewList as new ArrayList()
NewList = ArrayList.Repeat("test", 5)
```

ArrayList.Reverse()

Syntax

```
Void Reverse()  
Void Reverse( Int32 index, Int32 count )
```

Description

The `ArrayList.Reverse()` method reverses the order of all items in an array. The overloaded `Reverse()` method enables a developer to pass in an index and count to reverse only, a subset of the original arraylist.

Example

Listings 18.51 and 18.52 demonstrate how to build a simple arraylist and then reverse the order of the items.

Listing 18.51 Reversing the Order of Items in an Arraylist (C#)

```
ArrayList items = new ArrayList();  
items.Add("item1");  
items.Add("item2");  
  
items.Reverse();
```

Listing 18.52 Reversing the Order of Items in an Arraylist (Visual Basic.NET)

```
dim items = new ArrayList()  
items.Add("item1")  
items.Add("item2")  
  
items.Reverse()
```

ArrayList.SetRange()

Syntax

```
Void SetRange( Int32 index, ICollection c )
```

Description

The `ArrayList.SetRange()` method sets the range of a given arraylist to that of a collection passed in as an argument.

Example

Listings 18.53 and 18.54 demonstrate how to set the range of the items in an arraylist.

Listing 18.53 Setting the Range of an ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

ArrayList NewList = new ArrayList();
NewList.Add("newValue1");
NewList.Add("newValue2");

items.SetRange(1, NewList);
```

Listing 18.54 Setting the Range of an ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

dim NewList as new ArrayList()
NewList.Add("newValue1")
NewList.Add("newValue2")

items.SetRange(1, NewList)
```

ArrayList.Sort()

Syntax

```
Void Sort()
Void Sort( IComparer comparer )
Void Sort( Int32 index, Int32 count, IComparer comparer )
```

Description

The `ArrayList.Sort()` method enables a developer to sort all or part of an arraylist. The simplest `Sort()` method sorts the entire arraylist based on the list values. The first overloaded method enables the developer to pass in a comparer to better control how the sort is performed. The comparer must be of type `IComparer`.

The last overloaded method enables a developer to pass in an index and count in addition to the comparer in order to sort only a subset of the items in an arraylist.

Example

Listings 18.55 and 18.56 demonstrate how to perform a sort on a simple arraylist. When the sort is done executing, the items in the arraylist are ordered by number, in ascending order. If you want the sorted arraylist in descending order, one option is to sort the list and then use the `ArrayList.Reverse()` method.

Listing 18.55 Sorting ArrayList Items (C#)

```
ArrayList NewList = new ArrayList();
NewList.Add(5);
NewList.Add(6);
NewList.Add(3);

NewList.Sort();
```

Listing 18.56 Sorting ArrayList Items (Visual Basic.NET)

```
dim NewList as new ArrayList()
NewList.Add(5)
NewList.Add(6)
NewList.Add(3)

NewList.Sort()
```

ArrayList.Synchronized

Syntax

```
ArrayList Synchronized( ArrayList list )
IList Synchronized( IList list )
```

Description

The `ArrayList.Synchronized()` method accepts either an `ArrayList` or `IList` object and returns a synchronized list of the same type.

Example

Listings 18.57 and 18.58 demonstrate how to use the `Synchronized()` method to get a synchronized copy of a current `ArrayList`.

Listing 18.57 Synchronizing an ArrayList (C#)

```
ArrayList items = new ArrayList();
items.Add("item1");
items.Add("item2");

items = ArrayList.Synchronized(items);
```

Listing 18.58 Synchronizing an ArrayList (Visual Basic.NET)

```
dim items = new ArrayList()
items.Add("item1")
items.Add("item2")

items = ArrayList.Synchronized(items)
```

ArrayList.ToArray

Syntax

```
Object[] ToArray()  
Array ToArray( Type type )
```

Description

The `ArrayList.ToArray()` method returns an object array back to the caller, with items mirroring those in the original arraylist. If the type is passed into the call to `ToArrayList()`, then an array of that type is returned.

Example

Listings 18.59 and 18.60 demonstrate how to use `ArrayList.ToArray()` to create and return an object array.

Listing 18.59 Creating an Array from an ArrayList (C#)

```
Object[] MyArray = items.ToArray();
```

Listing 18.60 Creating an Array from an ArrayList (Visual Basic.NET)

```
dim MyArray as object()  
MyArray = items.ToArray()
```

ArrayList.TrimToSize

Syntax

```
Void TrimToSize()
```

Description

The `ArrayList.TrimToSize()` method sets `ArrayList.Capacity` to the number of elements in the arraylist.

Example

Listings 18.61 and 18.62 demonstrate how to use the `ArrayList.TrimToSize()` method to remove all unfilled entries in an arraylist.

Listing 18.61 Trimming an ArrayList (C#)

```
items.TrimToSize();
```

Listing 18.62 Trimming an ArrayList (Visual Basic.NET)

```
items.TrimToSize()
```

The BitArray Class

The `BitArray` class provides a number of properties and methods that you use to work with a bitarray. These properties and methods are listed in Table 18.2.

Table 18.2 Properties and Methods of the BitArray Class

Item	Description
Properties	
Count	Contains the number of items currently in the bitarray
IsReadOnly	Contains True if the bitarray is read-only and False otherwise
IsSynchronized	Contains True if the bitarray is synchronized and False otherwise
Item[]	Contains an index of all the items in the bitarray
Length	Enables the developer to get and change the length of the bitarray
Methods	
And()	Performs the logical And operation on the bitarray
Clone()	Returns a copy of the bitarray
CopyTo()	Copies the bitarray to another type of collection
Get()	Returns the value of a specific location in the bitarray
Not()	Performs the logical Not operation on the bitarray
Or()	Performs the logical or operation on the bitarray
Set()	Sets the value of a specific location in the bitarray
SetAll()	Sets all the values in the bitarray
Xor()	Performs the logical Xor operation on the bitarray

BitArray.Count

Syntax

Int32 Count

Description

The BitArray.Count property contains the number of elements in the bitarray.

Example

Listings 18.63 and 18.64 demonstrate how to use the Count property.

Listing 18.63 Using the BitArray.Count Property (C#)

```
BitArray ba = new BitArray(3);
ba.Set(0, false);
ba.Set(1, true);
ba.Set(2, false);

MyLabel.Text = "Number of members in BitArray: " + ba.Count.ToString();
```

Listing 18.64 Using the BitArray.Count Property (Visual Basic.NET)

```
dim ba as new BitArray(3)
ba.Set(0, false)
ba.Set(1, true)
```

Listing 18.64 continued

```
ba.Set(2, false)  
  
MyLabel.Text = "Number of members in BitArray: " + ba.Count.ToString()
```

BitArray.IsReadOnly

Syntax

```
Boolean IsReadOnly
```

Description

The `BitArray.IsReadOnly` contains a Boolean value that is used to determine the read-only status of the bitarray.

Example

Listings 18.65 and 18.66 demonstrate how to use the `BitArray.IsReadOnly` property.

Listing 18.65 Using the BitArray.IsReadOnly Property (C#)

```
BitArray ba = new BitArray(3);  
ba.Set(0, false);  
ba.Set(1, true);  
ba.Set(2, false);  
  
MyLabel.Text = "Is BitArray read-only?: " + ba.IsReadOnly.ToString();
```

Listing 18.66 Using the BitArray.IsReadOnly Property (Visual Basic.NET)

```
dim ba as new BitArray(3)  
ba.Set(0, false)  
ba.Set(1, true)  
ba.Set(2, false)  
  
MyLabel.Text = "Is BitArray read-only?: " + ba.IsReadOnly.ToString()
```

BitArray.IsSynchronized

Syntax

```
Boolean IsSynchronized
```

Description

The `BitArray.IsSynchronized` property contains a Boolean value that is used to determine whether the bitarray is synchronized.

Example

Listings 18.67 and 18.68 demonstrate how to use the `BitArray.IsSynchronized` property.

Listing 18.67 Using the BitArray.IsSynchronized Property (C#)

```
BitArray ba = new BitArray(3);
ba.Set(0, false);
ba.Set(1, true);
ba.Set(2, false);

MyLabel.Text = "Is BitArray synchronized?: " + ba.IsSynchronized.ToString();
```

Listing 18.68 Using the BitArray.IsSynchronized Property (Visual Basic.NET)

```
dim ba as new BitArray(3)
ba.Set(0, false)
ba.Set(1, true)
ba.Set(2, false)

MyLabel.Text = "Is BitArray synchronized?: " + ba.IsSynchronized.ToString()
```

BitArray.Item[]

Syntax

```
Boolean BitArray.Item[ Int32 index ]
```

Description

The BitArray.Item[] property is used to access individual elements of a bitarray.

Example

Listings 18.69 and 18.70 demonstrate how to use the BitArray.Item[] property.

Listing 18.69 Using the BitArray.Item[] Property (C#)

```
BitArray ba = new BitArray(3);
ba.Set(0, false);
ba.Set(1, true);
ba.Set(2, false);

MyLabel.Text = "Second element is " + ba[1].ToString();
```

Listing 18.70 Using the BitArray.Item() Property (Visual Basic.NET)

```
dim ba as new BitArray(3)
ba.Set(0, false)
ba.Set(1, true)
ba.Set(2, false)

MyLabel.Text = "Second element is " + ba(1).ToString()
```

BitArray.Length

Syntax

Int32 Length

Description

The `BitArray.Length` property can be used to both access and change the length of the bitarray.

Example

Listings 18.71 and 18.72 demonstrate how to use the `BitArray.Length` property. The first line of code in the example changes the length of the bitarray to 40. Then the `Length` property is used to display the bitarray length on the page.

Listing 18.71 Using the BitArray.Length Property (C#)

```
BitArray ba = new BitArray(3);
ba.Set(0, false);
ba.Set(1, true);
ba.Set(2, false);

ba.Length = 40;
MyLabel.Text = "Length of array: " + ba.Length.ToString();
```

Listing 18.72 Using the BitArray.Length Property (Visual Basic.NET)

```
dim ba as new BitArray(3)
ba.Set(0, false)
ba.Set(1, true)
ba.Set(2, false)

ba.Length = 40
MyLabel.Text = "Length of array: " + ba.Length.ToString()
```

BitArray.And()

Syntax

`BitArray And(BitArray value)`

Description

The `BitArray.And()` method is used to perform the `And` logical operation on a bitarray.

Example

Listings 18.73 and 18.74 demonstrate how to use the `BitArray.And()` method.

Listing 18.73 Using the BitArray.And() Method (C#)

```
BitArray ba = new BitArray(3);
ba.Set(0, false);
ba.Set(1, true);
ba.Set(2, false);

BitArray ba2 = new BitArray(3);

BitArray results = new BitArray(3);

results = ba2.And(ba);
```

Listing 18.74 Using the BitArray.And() Method (Visual Basic.NET)

```
dim ba as new BitArray(3)
ba.Set(0, false)
ba.Set(1, true)
ba.Set(2, false)

dim ba2 as new BitArray(3)

dim results as new BitArray(3)

results = ba2.And(ba)
```

BitArray.Clone()

Syntax

Object.Clone()

Description

The *BitArray.Clone()* method returns an object containing the bitarray on which this operation is being performed.

Example

Listings 18.75 and 18.76 demonstrate how to use the *BitArray.Clone()* method.

Listing 18.75 Using the BitArray.Clone() Method (C#)

```
BitArray ba = new BitArray(3);
ba.Set(0, false);
ba.Set(1, true);
ba.Set(2, false);

BitArray results = new BitArray(3);

results = (BitArray)ba.Clone();
```

Listing 18.76 Using the BitArray.Clone() Method (Visual Basic.NET)

```
dim ba as new BitArray(3)
ba.Set(0, false)
ba.Set(1, true)
ba.Set(2, false)

dim results as new BitArray(3)

results = ba.Clone()
```

BitArray.CopyTo()

Syntax

Void CopyTo(Array array, Int32 index)

Description

The `BitArray.CopyTo()` method copies a bitarray to an array passed in as the first argument, starting at an index provided by the second method argument.

Example

Listings 18.77 and 18.78 demonstrate how to use the `BitArray.CopyTo()` method.

Listing 18.77 Using the BitArray.CopyTo() Method (C#)

```
BitArray ba = new BitArray(3);
ba.Set(0, false);
ba.Set(1, true);
ba.Set(2, false);

Boolean[] MyArray = new Boolean[3];
ba.CopyTo(MyArray, 0);
```

Listing 18.78 Using the BitArray.CopyTo() Method (Visual Basic.NET)

```
dim ba as new BitArray(3)
ba.Set(0, false)
ba.Set(1, true)
ba.Set(2, false)

dim MyArray(3) as Boolean
ba.CopyTo(MyArray, 0)
```

BitArray.Get()

Syntax

Boolean Get(Int32 index)

Description

The `BitArray.Get()` method returns the value of a specific element of a bitarray.

Example

Listings 18.79 and 18.80 demonstrate how to use the `BitArray.Get()` method.

Listing 18.79 Using the BitArray.Get() Method (C#)

```
BitArray ba = new BitArray(3);
ba.Set(0, false);
ba.Set(1, true);
ba.Set(2, false);

Boolean MyBool;
MyBool = ba.Get(0);
MyLabel.Text = "MyBool is " + MyBool.ToString();
```

Listing 18.80 Using the BitArray.Get() Method (Visual Basic.NET)

```
dim ba as new BitArray(3)
ba.Set(0, false)
ba.Set(1, true)
ba.Set(2, false)

dim MyBool as Boolean
MyBool = ba.Get(0)
MyLabel.Text = "MyBool is " + MyBool.ToString()
```

BitArray.Not()

Syntax

`BitArray Not()`

Description

The `BitArray.Not()` method performs the `Not` logical operation on a bitarray.

Example

Listings 18.81 and 18.82 demonstrate how to use the `BitArray.Not()` method.

Listing 18.81 Using the BitArray.Not() Method (C#)

```
BitArray ba = new BitArray(3);
ba.Set(0, false);
ba.Set(1, true);
ba.Set(2, false);

BitArray results = new BitArray(3);
```

Listing 18.81 continued

```
results = ba.Not();
MyList2.DataSource = results;
MyList2.DataBind();
```

Listing 18.82 Using the BitArray.Not() Method (Visual Basic.NET)

```
dim ba as new BitArray(3)
ba.Set(0, false)
ba.Set(1, true)
ba.Set(2, false)

dim results as new BitArray(3)

results = ba.Not()
MyList2.DataSource = results
MyList2.DataBind()
```

BitArray.Or()

Syntax

BitArray Or(BitArray value)

Description

The `BitArray.Or()` method uses the `value` argument to perform the `Or` logical operation on a bitarray.

Example

Listings 18.83 and 18.84 demonstrate how to use the `BitArray.Or()` method.

Listing 18.83 Using the BitArray.Or() Method (C#)

```
BitArray ba = new BitArray(3);
ba.Set(0, false);
ba.Set(1, true);
ba.Set(2, false);

BitArray ba2 = new BitArray(3);

ba.Or(ba2);
```

Listing 18.84 Using the BitArray.Or() Method (Visual Basic.NET)

```
dim ba as new BitArray(3)
ba.Set(0, false)
ba.Set(1, true)
ba.Set(2, false)
```

Listing 18.84 continued

```
dim ba2 as new BitArray(3)  
  
ba.Or(ba2)
```

BitArray.Set()

Syntax

```
Void Set( Int32 index, Boolean value )
```

Description

The `BitArray.Set()` method sets an element of the bitarray that is given by the first argument to the value provided in the second argument.

Example

Listings 18.85 and 18.86 demonstrate how to use the `BitArray.Set()` method.

Listing 18.85 Using the BitArray.Set() Method (C#)

```
BitArray ba2 = new BitArray(3);  
ba2.Set(0, true);  
ba2.Set(1, true);  
ba2.Set(2, false);
```

Listing 18.86 Using the BitArray.Set() Method (Visual Basic.NET)

```
dim ba as new BitArray(3)  
ba.Set(0, false)  
ba.Set(1, true)  
ba.Set(2, false)
```

BitArray.SetAll()

Syntax

```
Void SetAll( Boolean value )
```

Description

The `BitArray.SetAll()` method sets all elements of the bitarray to a value provided as the argument to the method.

Example

Listings 18.87 and 18.88 demonstrate how to use the `BitArray.SetAll()` method.

Listing 18.87 Using the BitArray.SetAll() Method (C#)

```
BitArray ba = new BitArray(3);  
  
ba.SetAll(true);
```

Listing 18.88 Using the BitArray.SetAll() Method (Visual Basic.NET)

```
dim ba as new BitArray(3)  
  
ba.SetAll(true)
```

BitArray.Xor()

Syntax

```
BitArray Xor( BitArray value )
```

Description

The `BitArray.Xor()` method performs the `Xor` logical operation on a bitarray, using the value that the bitarray passed in as the argument to the method.

Example

Listings 18.89 and 18.90 demonstrate how to use the `BitArray.Xor()` method.

Listing 18.89 Using the BitArray.Xor() Method (C#)

```
BitArray ba = new BitArray(3);  
ba.Set(0, false);  
ba.Set(1, true);  
ba.Set(2, false);  
  
BitArray ba2 = new BitArray(3);  
  
ba.Xor(ba2);
```

Listing 18.90 Using the BitArray.Xor() Method (Visual Basic.NET)

```
dim ba as new BitArray(3)  
ba.Set(0, false)  
ba.Set(1, true)  
ba.Set(2, false)  
  
dim ba2 as new BitArray(3)  
  
ba.Xor(ba2)
```

The Hashtable Class

The `Hashtable` class provides a number of properties and methods that you use when working with a hashtable. These properties and methods are listed in Table 18.3.

Table 18.3 Properties and Methods of the Hashtable Class

Item	Description
Properties	
Count	Contains the number of items currently in the hashtable
IsReadOnly	Contains True if the hashtable is read-only and False otherwise
IsSynchronized	Contains True if the hashtable is synchronized and False otherwise
Item[]	Contains an index of all the items in the hashtable
Keys	Contains a collection of all the keys in the hashtable
Values	Contains a collection of all the values in the hashtable
Methods	
Add()	Adds an item to the hashtable
Clear()	Clears all items from the hashtable
Clone()	Returns a duplicate of the hashtable
Contains()	Returns True if a given object is in the current hashtable
ContainsKey()	Returns True if a given key is in the current hashtable
ContainsValue()	Returns True if a given value is in the current hashtable
CopyTo()	Copies all or part of the current hashtable to another hashtable that is passed in as an argument
Remove()	Removes a given item from the hashtable
Synchronized()	Synchronizes the hashtable

Hashtable.Count

Syntax

Int32 Count

Description

The `Hashtable.Count` property contains the total number of objects in the hashtable.

Example

Listings 18.91 and 18.92 demonstrate how to use the `Hashtable.Count` property.

Listing 18.91 Using the `Hashtable.Count` Property (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

MyLabel.Text = "Count is " + hTable.Count.ToString();
```

Listing 18.92 Using the Hashtable.Count Property (Visual Basic.NET)

```
dim hTable as new Hashtable()  
hTable.Add("key1", "value1")  
hTable.Add("key2", "value2")  
hTable.Add("key3", "value3")  
  
MyLabel.Text = "Count is " + hTable.Count.ToString()
```

Hashtable.IsReadOnly

Syntax

Boolean IsReadOnly

Description

The `Hashtable.IsReadOnly` property enables a developer to check the read-only status of a hashtable.

Example

Listings 18.92 and 18.93 demonstrate how to use the `Hashtable.IsReadOnly` property.

Listing 18.92 Using the Hashtable.IsReadOnly Property (C#)

```
Hashtable hTable = new Hashtable();  
hTable.Add("key1", "value1");  
hTable.Add("key2", "value2");  
hTable.Add("key3", "value3");  
  
MyLabel.Text = "IsReadOnly set to " + hTable.IsReadOnly;
```

Listing 18.93 Using the Hashtable.IsReadOnly Property (Visual Basic.NET)

```
dim hTable as new Hashtable()  
hTable.Add("key1", "value1")  
hTable.Add("key2", "value2")  
hTable.Add("key3", "value3")  
  
MyLabel.Text = "IsReadOnly set to " + hTable.IsReadOnly.ToString()
```

Hashtable.IsSynchronized

Syntax

Boolean IsSynchronized

Description

The `Hashtable.IsSynchronized` property enables a developer to see if a given hashtable is synchronized.

Example

Listings 18.94 and 18.95 demonstrate how to use the `Hashtable.IsSynchronized` property.

Listing 18.94 Using the `Hashtable.IsSynchronized` Property (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

MyLabel.Text = "IsSynchronized set to " + hTable.IsSynchronized;
```

Listing 18.95 Using the `Hashtable.IsSynchronized` Property (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

MyLabel.Text = "IsSynchronized set to " + hTable.IsSynchronized.ToString()
```

Hashtable.Item

Syntax

Object `Item(Object key)`

Description

The `Hashtable.Item` property enables a developer to access individual elements of a hashtable.

Example

Listings 18.96 and 18.97 demonstrate how to use the `Hashtable.Item` property.

Listing 18.96 Using the `Hashtable.Item` Property (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

MyLabel.Text = "Second Item in hTable: " + (string)hTable["key2"];
```

Listing 18.97 Using the `Hashtable.Item()` Property (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
```

Listing 18.97 continued

```
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

MyLabel.Text = "Second Item in hTable: " + hTable("key2")
```

Hashtable.Keys

Syntax

ICollection Keys

Description

The `Hashtable.Keys` property contains a collection of all keys in a hashtable.

Example

Listings 18.98 and 18.99 demonstrate how to use the `Hashtable.Keys` property.

Listing 18.98 Using the Hashtable.Keys Property (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

MyList.DataSource = hTable.Keys;
MyList.DataBind();
```

Listing 18.99 Using the Hashtable.Keys Property (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

MyList.DataSource = hTable.Keys
MyList.DataBind()
```

Hashtable.Values

Syntax

ICollection Values

Description

The `Hashtable.Values` property contains a collection of all values in a hashtable.

Example

Listings 18.100 and 18.101 demonstrate how to use the `Hashtable.Values` property.

Listing 18.100 Using the Hashtable.Values Property (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

MyList2.DataSource = hTable.Values;
MyList2.DataBind();
```

Listing 18.101 Using the Hashtable.Values Property (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

MyList2.DataSource = hTable.Values
MyList2.DataBind()
```

Hashtable.Add()

Syntax

```
Void Add( Object key, Object value )
```

Description

The `Hashtable.Add()` method adds name/value pairs to the hashtable.

Example

Listings 18.102 and 18.103 demonstrate how to use the `Hashtable.Add()` method.

Listing 18.102 Using the Hashtable.Add() Method (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");
```

Listing 18.103 Using the Hashtable.Add() Method (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")
```

Hashtable.Clear()

Syntax

```
Void Clear()
```

Description

The `Hashtable.Clear()` method clears all keys and values from the hashtable.

Example

Listings 18.104 and 18.105 demonstrate how to use the `Hashtable.Clear()` method.

Listing 18.104 Using the Hashtable.Clear() Method (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

hTable.Clear();
```

Listing 18.105 Using the Hashtable.Clear() Method (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

hTable.Clear()
```

Hashtable.Clone()

Syntax

```
Object Clone()
```

Description

The `Hashtable.Clone()` method creates an identical new hashtable.

Example

Listings 18.106 and 18.107 demonstrate how to use the `Hashtable.Clone()` method.

Listing 18.106 Using the Hashtable.Clone() Method (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

Hashtable hTable3 = (Hashtable) hTable.Clone();
```

Listing 18.107 Using the Hashtable.Clone() Method (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
```

Listing 18.107 continued

```

hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

dim hTable3 as new Hashtable()
hTable3 = hTable.Clone()

```

Hashtable.Contains()

Syntax

Boolean Contains(Object key)

Description

The `Hashtable.Contains()` method enables a developer to check for the existence of a particular key in a hashtable.

Example

Listings 18.108 and 18.109 demonstrate how to use the `Hashtable.Contains()` method.

Listing 18.108 Using the Hashtable.Contains() Method (C#)

```

Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

MyLabel.Text = "Does hTable contain this key: "
              + hTable.Contains("key1").ToString();

```

Listing 18.109 Using the Hashtable.Contains() Method (Visual Basic.NET)

```

dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

MyLabel.Text = "Does hTable contain this key: "
              + hTable.Contains("key1").ToString()

```

Hashtable.ContainsKey()

Syntax

Boolean ContainsKey(Object key)

Description

The `Hashtable.ContainsKey()` method enables a developer to check for the existence of a particular key in a hashtable.

Example

Listings 18.110 and 18.111 demonstrate how to use the `Hashtable.ContainsKey()` method.

Listing 18.110 Using the `Hashtable.ContainsKey()` Method (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

MyLabel.Text = "Does hTable contain this key: "
    + hTable.ContainsKey("key2").ToString();
```

Listing 18.111 Using the `Hashtable.ContainsKey()` Method (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

MyLabel.Text = "Does hTable contain this key: "
    + hTable.ContainsKey("key2").ToString()
```

Hashtable.ContainsValue()

Syntax

```
Boolean ContainsValue( Object value )
```

Description

The `Hashtable.ContainsValue()` method enables a developer to check for the existence of a particular value in a hashtable.

Example

Listings 18.112 and 18.113 demonstrate how to use the `Hashtable.ContainsValue()` method.

Listing 18.112 Using the `Hashtable.ContainsValue()` Method (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");
```

Listing 18.112 continued

```
MyLabel.Text = "Does hTable contain this value: "
    + hTable.ContainsValue("value1").ToString();
```

Listing 18.113 Using the Hashtable.ContainsValue() Method (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

MyLabel.Text = "Does hTable contain this value: "
    + hTable.ContainsValue("value1").ToString()
```

Hashtable.CopyTo()

Syntax

```
Void CopyTo( Array array, Int32 arrayIndex )
```

Description

The `Hashtable.CopyTo()` method copies the hashtable to an array.

Example

Listings 18.114 and 18.115 demonstrate how to use the `Hashtable.CopyTo()` method.

Listing 18.114 Using the Hashtable.CopyTo() Method (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

String[] MyArray = new String[3];
hTable.CopyTo(MyArray, 0);
MyLabel.Text = "Third item of ArrayList is " + MyArray[2];
```

Listing 18.115 Using the Hashtable.CopyTo() Method (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

dim MyArray(3) as String
hTable.CopyTo(MyArray, 0)
MyLabel.Text = "Third item of ArrayList is " + MyArray(2)
```

Hashtable.Remove()

Syntax

```
Void Remove( Object key )
```

Description

The `Hashtable.Remove()` method removes an entry from the hashtable by key.

Example

Listings 18.116 and 18.117 demonstrate how to use the `Hashtable.Remove()` method.

Listing 18.116 Using the Hashtable.Remove() Method (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
hTable.Add("key3", "value3");

hTable.Remove("key3");
```

Listing 18.117 Using the Hashtable.Remove() Method (Visual Basic.NET)

```
dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

hTable.Remove("key3")
```

Hashtable.Synchronized()

Syntax

```
Hashtable Synchronized( Hashtable table )
```

Description

The `Hashtable.Synchronized()` method returns a synchronized duplicate of the current hashtable.

Example

Listings 18.118 and 18.119 demonstrate how to use the `Hashtable.Synchronized()` method.

Listing 18.118 Using the Hashtable.Synchronized() Method (C#)

```
Hashtable hTable = new Hashtable();
hTable.Add("key1", "value1");
hTable.Add("key2", "value2");
```

Listing 18.118 continued

```

hTable.Add("key3", "value3");

hTable = ArrayList.Synchronized(hTable);
MyLabel.Text = "True if items is synchronized: " + hTable.IsSynchronized;

```

Listing 18.119 Using the Hashtable.Synchronized() Method (Visual Basic.NET)

```

dim hTable as new Hashtable()
hTable.Add("key1", "value1")
hTable.Add("key2", "value2")
hTable.Add("key3", "value3")

hTable = Hashtable.Synchronized(hTable)
MyLabel.Text = "True if items is synchronized: " _
    + hTable.IsSynchronized.ToString()

```

The Queue Class

The Queue class provides a number of properties and methods that are used to work with a *queue*, which is an abstract data structure that uses a first in, first out (FIFO) structure. A queue operates much like a line at a supermarket. The Queue class properties and methods are listed in Table 18.4.

Table 18.4 Properties and Methods of the Queue Class

Item	Description
Properties	
Count	Contains the number of items currently in the queue
IsReadOnly	Contains True if the queue is read-only and False otherwise
IsSynchronized	Contains True if the queue is synchronized and False otherwise
Methods	
Clear()	Clears all items from the queue
Clone()	Creates a duplicate of the queue
Contains()	Returns True if a given object is in the current queue
CopyTo()	Copies all or part of the current queue to another queue that is passed in as an argument
Dequeue()	Returns the next item from the queue and then removes it from the queue
Enqueue()	Adds a new object to the end of the queue
Peek()	Returns the next item from the queue but does not remove it
Synchronized()	Synchronizes the queue
ToArray()	Copies all or part of the current queue to an array

Queue.Count

Syntax

Int32 Count

Description

The Queue.Count property contains the number of objects currently in the queue.

Example

Listings 18.120 and 18.121 demonstrate how to use the Queue.Count property.

Listing 18.120 Using the Queue.Count Property (C#)

```
Queue queue1 = new Queue();
queue1.Enqueue("item1");
queue1.Enqueue("item2");
queue1.Enqueue("item3");

MyLabel.Text = "Count is " + queue1.Count.ToString();
```

Listing 18.121 Using the Queue.Count Property (Visual Basic.NET)

```
dim queue1 as new Queue()
queue1.Enqueue("item1")
queue1.Enqueue("item2")
queue1.Enqueue("item3")

MyLabel.Text = "Count is " + queue1.Count.ToString()
```

Queue.IsReadOnly

Syntax

Boolean IsReadOnly

Description

The Queue.IsReadOnly property enables a developer to check the read-only status of a queue.

Example

Listings 18.122 and 18.123 demonstrate how to use the Queue.IsReadOnly property.

Listing 18.122 Using the Queue.IsReadOnly Property (C#)

```
Queue queue1 = new Queue();
queue1.Enqueue("item1");
queue1.Enqueue("item2");
queue1.Enqueue("item3");

MyLabel.Text = "IsReadOnly set to " + queue1.IsReadOnly;
```

Listing 18.123 Using the Queue.IsReadOnly Property (Visual Basic.NET)

```
dim queue1 as new Queue()  
queue1.Enqueue("item1")  
queue1.Enqueue("item2")  
queue1.Enqueue("item3")  
  
MyLabel.Text = "IsReadOnly set to " + queue1.IsReadOnly.ToString()
```

Queue.IsSynchronized

Syntax

```
Boolean IsSynchronized
```

Description

The Queue.IsSynchronized property enables a developer to check the synchronization of a queue.

Example

Listings 18.124 and 18.125 demonstrate how to use the Queue.IsSynchronized property.

Listing 18.124 Using the Queue.IsSynchronized Property (C#)

```
Queue queue1 = new Queue();  
queue1.Enqueue("item1");  
queue1.Enqueue("item2");  
queue1.Enqueue("item3");  
  
MyLabel.Text = "IsSynchronized set to " + queue1.IsSynchronized;
```

Listing 18.125 Using the Queue.IsSynchronized Property (Visual Basic.NET)

```
dim queue1 as new Queue()  
queue1.Enqueue("item1")  
queue1.Enqueue("item2")  
queue1.Enqueue("item3")  
  
MyLabel.Text = "IsSynchronized set to " + queue1.IsSynchronized.ToString()
```

Queue.Clear()

Syntax

```
Void Clear()
```

Description

The Queue.Clear() method removes all items from the queue.

Example

Listings 18.126 and 18.127 demonstrate how to use the Queue.Clear() method.

Listing 18.126 Using the Queue.Clear() Method (C#)

```
Queue queue1 = new Queue();
queue1.Enqueue("item1");
queue1.Enqueue("item2");
queue1.Enqueue("item3");

queue1.Clear();
```

Listing 18.127 Using the Queue.Clear() Method (Visual Basic.NET)

```
dim queue1 as new Queue()
queue1.Enqueue("item1")
queue1.Enqueue("item2")
queue1.Enqueue("item3")

queue1.Clear()
```

Queue.Clone()

Syntax

Object Clone()

Description

The Queue.Clone() method returns a queue that is identical to the one being cloned.

Example

Listings 18.128 and 18.129 demonstrate how to use the Queue.Clone() method. Notice that because the Queue.Clone() method returns an object, it must be cast to type Queue before it is assigned to queue3. Visual Basic.NET handles this typecasting implicitly.

Listing 18.128 Using the Queue.Clone() Method (C#)

```
Queue queue2 = new Queue();
queue2.Enqueue("item1");
queue2.Enqueue("item2");
queue2.Enqueue("item3");

Queue queue3 = (Queue) queue2.Clone();
```

Listing 18.129 Using the Queue.Clone() Method (Visual Basic.NET)

```
dim queue2 as new Queue()
queue2.Enqueue("item1")
queue2.Enqueue("item2")
```

Listing 18.129 continued

```
queue2.Enqueue("item3")

dim queue3 as new Queue()
queue3 = queue2.Clone()
```

Queue.Contains()

Syntax

Boolean Contains(Object obj)

Description

The Queue.Contains() method returns True if the argument obj is found in the queue.

Example

Listings 18.130 and 18.131 demonstrate how to use the Queue.Contains() method.

Listing 18.130 Using the Queue.Contains() Method (C#)

```
Queue queue1 = new Queue();
queue1.Enqueue("item1");
queue1.Enqueue("item2");
queue1.Enqueue("item3");

MyLabel.Text = "Does queue contain this object: "
    + queue1.Contains("item2").ToString();
```

Listing 18.131 Using the Queue.Contains() Method (Visual Basic.NET)

```
dim queue1 as new Queue()
queue1.Enqueue("item1")
queue1.Enqueue("item2")
queue1.Enqueue("item3")

MyLabel.Text = "Does queue contain this object: " _
    + queue1.Contains("item2").ToString()
```

Queue.CopyTo()

Syntax

Void CopyTo(Array array, Int32 index)

Description

The Queue.CopyTo() method copies a queue to an array.

Example

Listings 18.132 and 18.133 demonstrate how to use the Queue.CopyTo() method.

Listing 18.132 Using the Queue.CopyTo() Method (C#)

```
Queue queue1 = new Queue();
queue1.Enqueue("item1");
queue1.Enqueue("item2");
queue1.Enqueue("item3");

String[] MyArray = new String[3];
queue1.CopyTo(MyArray, 0);
MyLabel.Text = "Third item of queue1 is " + MyArray[2];
```

Listing 18.133 Using the Queue.CopyTo() Method (Visual Basic.NET)

```
dim queue1 as new Queue()
queue1.Enqueue("item1")
queue1.Enqueue("item2")
queue1.Enqueue("item3")

dim MyArray(3) as String
queue1.CopyTo(MyArray, 0)
MyLabel.Text = "Third item of queue1 is " + MyArray(2)
```

Queue.Dequeue()

Syntax

```
Object Dequeue()
```

Description

The Queue.Dequeue() method returns the object at the front of the queue and then removes it from the queue. If the queue is empty, InvalidOperationException is thrown at runtime.

Example

Listings 18.134 and 18.135 demonstrate how to use the Queue.Dequeue() method.

Listing 18.134 Using the Queue.Dequeue() Method (C#)

```
Queue queue1 = new Queue();
queue1.Enqueue("item1");
queue1.Enqueue("item2");
queue1.Enqueue("item3");

queue1.Dequeue();
```

Listing 18.135 Using the Queue.Dequeue() Method (Visual Basic.NET)

```
dim queue1 as new Queue()
queue1.Enqueue("item1")
queue1.Enqueue("item2")
```

Listing 18.135 continued

```
queue1.Enqueue("item3")
queue1.Dequeue()
```

Queue.Enqueue()**Syntax**

Void Enqueue(Object obj)

Description

The Queue.Enqueue() method puts an object or a value at the end of the queue.

Example

Listings 18.136 and 18.137 demonstrate how to use the Queue.Enqueue() method.

Listing 18.136 Using the Queue.Enqueue() Method (C#)

```
Queue queue1 = new Queue();
queue1.Enqueue(42);
```

Listing 18.137 Using the Queue.Enqueue() Method (Visual Basic.NET)

```
dim queue1 as new Queue()
queue1.Enqueue(42)
```

Queue.Peek()**Syntax**

Object Peek()

Description

The Queue.Peek() method returns the object to the front of the queue, but it does not remove this object from the queue.

Example

Listings 18.138 and 18.139 demonstrate how to use the Queue.Peek() method.

Listing 18.138 Using the Queue.Peek() Method (C#)

```
Queue queue1 = new Queue();
queue1.Enqueue("item1");
queue1.Enqueue("item2");
queue1.Enqueue("item3");
```

```
MyLabel.Text = "The next object in queue1 is '" + queue1.Peek() + "'";
```

Listing 18.139 Using the Queue.Peek() Method (Visual Basic.NET)

```
dim queue1 as new Queue()  
queue1.Enqueue("item1")  
queue1.Enqueue("item2")  
queue1.Enqueue("item3")  
  
MyLabel.Text = "The next object in queue1 is '" + queue1.Peek() + "'"
```

Queue.Synchronized()

Syntax

```
Queue Synchronized( Queue queue )
```

Description

The Queue.Synchronized() method returns a synchronized copy of the current queue.

Example

Listings 18.140 and 18.141 demonstrate how to use the Queue.Synchronized() method.

Listing 18.140 Using the Queue.Synchronized() Method (C#)

```
Queue queue1 = new Queue();  
queue1.Enqueue("item1");  
queue1.Enqueue("item2");  
queue1.Enqueue("item3");  
  
queue1 = Queue.Synchronized(queue1);  
MyLabel.Text = "True if items is synchronized: " + queue1.IsSynchronized;
```

Listing 18.141 Using the Queue.Synchronized() Method (Visual Basic.NET)

```
dim queue1 as new Queue()  
queue1.Enqueue("item1")  
queue1.Enqueue("item2")  
queue1.Enqueue("item3")  
  
queue1 = Queue.Synchronized(queue1)  
MyLabel.Text = "True if items is synchronized:  
+ queue1.IsSynchronized.ToString()
```

Queue.ToArray()

Syntax

```
Object[] ToArray()
```

Description

The Queue.ToArray() method copies the current queue to an object array.

Example

Listings 18.142 and 18.143 demonstrate how to use the Queue.ToArray() method.

Listing 18.142 Using the Queue.ToArray() Method (C#)

```
Queue queue1 = new Queue();
queue1.Enqueue("item1");
queue1.Enqueue("item2");
queue1.Enqueue("item3");

Object[] MyArray;
MyArray = queue1.ToArray();
```

Listing 18.143 Using the Queue.ToArray() Method (Visual Basic.NET)

```
dim queue1 as new Queue()
queue1.Enqueue("item1")
queue1.Enqueue("item2")
queue1.Enqueue("item3")

dim MyArray() as Object
MyArray = queue1.ToArray()
```

The SortedList Class

The SortedList class provides a number of properties and methods that you use to work with a sortedlist. The properties and methods are listed in Table 18.5.

Table 18.5 Properties and Methods of the SortedList Class

Item	Description
Properties	
Capacity	Contains the allocated length of the sortedlist
Count	Contains the number of items currently in the sortedlist
IsReadOnly	Contains True if the sortedlist is read-only and False otherwise
IsSynchronized	Contains True if the sortedlist is synchronized
Item[]	Contains an index of all the items in the sortedlist
Keys	Contains a collection of all keys in the sortedlist
Values	Contains a collection of all values in the sortedlist
Methods	
Add()	Adds an item to the sortedlist and returns the newly added index
Clear()	Clears all items from the sortedlist
Clone()	Creates a duplicate of the sortedlist

Table 18.5 continued

Item	Description
Methods	
Contains()	Returns True if a given object is in the current sortedlist
ContainsKey()	Returns a collection of all keys in the sortedlist
ContainsValue()	Returns a collection of all values in the sortedlist
CopyTo()	Copies all or part of the current sortedlist to another sortedlist
GetByIndex()	Returns the value of a specific location in the sortedlist
GetKey()	Returns the key of a specific location in the sortedlist
GetKeyList()	Returns the collection of keys in the sortedlist
GetValueList()	Returns the collection of values in the sortedlist
IndexOfKey()	Returns the index of a specific key
IndexOfValue()	Returns the index of a specific value
Remove()	Removes an item from the sortedlist by key
RemoveAt()	Removes an item from the sortedlist by index
SetByIndex()	Sets the value of an item in the sortedlist by index
Synchronized()	Returns a synchronized copy of the sortedlist
TrimToSize()	Changes the size of the sortedlist to match the number of items currently in the sortedlist

SortedDictionary.Capacity

Syntax

Int32.Capacity

Description

The `SortedDictionary.Capacity` property contains the maximum size of the sortedlist. The default capacity of a sortedlist is 16.

Example

Listings 18.144 and 18.145 demonstrate how to use the `SortedDictionary.Capacity` property.

Listing 18.144 Using the SortedDictionary.Capacity Property (C#)

```
SortedDictionary items = new SortedDictionary();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

items.Capacity = 5;
MyLabel.Text = "Capacity is " + items.Capacity.ToString();
```

Listing 18.145 Using the SortedList.Capacity Property (Visual Basic.NET)

```
dim items as new SortedList()  
items.Add("key2", 6)  
items.Add("key3", 1)  
items.Add("key1", 4)  
  
items.Capacity = 5  
MyLabel.Text = "Capacity is " + items.Capacity.ToString()
```

SortedList.Count

Syntax

Int32 Count

Description

The SortedList.Count property contains the current number of items in the sortedlist.

Example

Listings 18.146 and 18.147 demonstrate how to use the SortedList.Count property.

Listing 18.146 Using the SortedList.Count Property (C#)

```
SortedList items = new SortedList();  
items.Add("key2", 6);  
items.Add("key3", 1);  
items.Add("key1", 4);  
  
MyLabel.Text = "Count is " + items.Count.ToString();
```

Listing 18.147 Using the SortedList.Count Property (Visual Basic.NET)

```
dim items as new SortedList()  
items.Add("key2", 6)  
items.Add("key3", 1)  
items.Add("key1", 4)  
  
MyLabel.Text = "Count is " + items.Count.ToString()
```

SortedList.IsReadOnly

Syntax

Boolean IsReadOnly

Description

The SortedList.IsReadOnly property returns true if the current sortedlist is read-only.

Example

Listings 18.148 and 18.149 demonstrate how to use the `SortedList.IsReadOnly` property.

Listing 18.148 Using the `SortedList.IsReadOnly` Property (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyLabel.Text = "IsReadOnly set to " + items.IsReadOnly;
```

Listing 18.149 Using the `SortedList.IsReadOnly` Property (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

MyLabel.Text = "IsReadOnly set to " + items.IsReadOnly.ToString()
```

SortedList.IsSynchronized

Syntax

Boolean IsSynchronized

Description

The `SortedList.IsSynchronized` property is True if the current sortedlist is synchronized.

Example

Listings 18.150 and 18.151 demonstrate how to use the `SortedList.IsSynchronized` property.

Listing 18.150 Using the `SortedList.IsSynchronized` Property (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyLabel.Text = "IsSynchronized set to " + items.IsSynchronized;
```

Listing 18.151 Using the SortedList.IsSynchronized Property (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

MyLabel.Text = "IsSynchronized set to " + items.IsSynchronized.ToString()
```

SortedList.Item

Syntax

Object Item(*Object* key)

Description

The SortedList.Item property enables a developer to retrieve a value by using a key as an index.

Example

Listings 18.152 and 18.153 demonstrate how to use the SortedList.Item property.

Listing 18.152 Using the SortedList.Item Property (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyLabel.Text = "Second Item in SortedList: " + items["key1"];
```

Listing 18.153 Using the SortedList.Item[] Property (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

MyLabel.Text = "Second Item in SortedList: " + items("key1").ToString()
```

SortedList.Keys

Syntax

ICollection Keys

Description

The SortedList.Keys property contains a collection of keys in the sortedlist.

Example

Listings 18.154 and 18.155 demonstrate how to use the `SortedList.Keys` property.

Listing 18.154 Using the `SortedList.Keys` Property (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyList.DataSource = items.Keys;
MyList.DataBind();
```

Listing 18.155 Using the `SortedList.Keys` Property (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

MyList.DataSource = items.Keys
MyList.DataBind()
```

SortedList.Values

Syntax

ICollection Values

Description

The `SortedList.Values` property contains a collection of values in the sortedlist.

Example

Listings 18.156 and 18.157 demonstrate how to use the `SortedList.Values` property.

Listing 18.156 Using the `SortedList.Values` Property (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyList.DataSource = items.Values;
MyList.DataBind();
```

Listing 18.157 Using the `SortedList.Values` Property (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
```

Listing 18.157 continued

```
items.Add("key1", 4)

MyList.DataSource = items.Values
MyList.DataBind()
```

SortedList.Add()

Syntax

```
Void Add( Object key, Object value )
```

Description

The SortedList.Add() method adds key/value pairs to the sortedlist.

Example

Listings 18.158 and 18.159 demonstrate how to use the SortedList.Add() method.

Listing 18.158 Using the SortedList.Add() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);
```

Listing 18.159 Using the SortedList.Add() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)
```

SortedList.Clear()

Syntax

```
Void Clear()
```

Description

The SortedList.Clear() method removes all keys and values from the current sortedlist.

Example

Listings 18.160 and 18.161 demonstrate how to use the SortedList.Clear() method.

Listing 18.160 Using the SortedList.Clear() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
```

Listing 18.160 continued

```
items.Add("key3", 1);
items.Add("key1", 4);

items.Clear();
```

Listing 18.161 Using the SortedList.Clear() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

items.Clear()
```

SortedList.Clone()

Syntax

Object **Clone()**

Description

The `SortedList.Clone()` method returns a copy of the current `sortedlist`.

Example

Listings 18.162 and 18.163 demonstrate how to use the `SortedList.Clone()` method.

Listing 18.162 Using the SortedList.Clone() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

ArrayList NewList = (ArrayList) items.Clone();
```

Listing 18.163 Using the SortedList.Clone() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

dim NewList as new SortedList()
NewList = items.Clone()
```

SortedList.Contains()

Syntax

Boolean **Contains(Object key)**

Description

The `SortedList.Contains()` method returns True if the current sortedlist contains the key.

Example

Listings 18.164 and 18.165 demonstrate how to use the `SortedList.Contains()` method.

Listing 18.164 Using the SortedList.Contains() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyLabel.Text = "Does list contain this key: "
    + items.Contains("key2").ToString();
```

Listing 18.165 Using the SortedList.Contains() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

MyLabel.Text = "Does list contain this key: " _
    + items.Contains("key2").ToString()
```

SortedList.ContainsKey()

Syntax

```
Boolean ContainsKey( Object key )
```

Description

The `SortedList.ContainsKey()` method returns True if the current sortedlist contains the key.

Example

Listings 18.166 and 18.167 demonstrate how to use the `SortedList.ContainsKey()` method.

Listing 18.166 Using the SortedList.ContainsKey() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);
```

Listing 18.166 continued

```
MyLabel.Text = "Does list contain this key: "
+ items.Contains("key3").ToString();
```

Listing 18.167 Using the SortedList.ContainsKey() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)
```

```
MyLabel.Text = "Does list contain this key: "
+ items.Contains("key3").ToString()
```

SortedList.ContainsValue()

Syntax

```
Boolean ContainsValue( Object value )
```

Description

The `SortedList.ContainsValue()` method returns True if the current sortedlist contains the value.

Example

Listings 18.168 and 18.169 demonstrate how to use the `SortedList.ContainsValue()` method.

Listing 18.168 Using the SortedList.ContainsValue() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyLabel.Text = "Does list contain this value: "
+ items.Contains("1").ToString();
```

Listing 18.169 Using the SortedList.ContainsValue() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

MyLabel.Text = "Does list contain this value: "
+ items.Contains("1").ToString()
```

SortedList.CopyTo()

Syntax

```
Void CopyTo( Array array, Int32 index )
```

Description

The `SortedList.CopyTo()` method copies the current `SortedList` to an array.

Example

Listings 18.170 and 18.171 demonstrate how to use the `SortedList.CopyTo()` method.

Listing 18.170 Using the SortedList.CopyTo() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

String[] MyArray = new String[4];
items.CopyTo(MyArray);
MyLabel.Text = "Third item of SortedList is " + MyArray[2];
```

Listing 18.171 Using the SortedList.CopyTo() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

dim MyArray(4) as String
items.CopyTo(MyArray, 0)
MyLabel.Text = "Third item of SortedList is " + MyArray(2)
```

SortedList.GetByIndex()

Syntax

```
Object GetByIndex( Int32 index )
```

Description

The `SortedList.GetByIndex()` method returns the value of a particular location in the list, given by the `index` argument.

Example

Listings 18.172 and 18.173 demonstrate how to use the `SortedList.GetByIndex()` method.

Listing 18.172 Using the SortedList.GetByIndex() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyLabel.Text = "Value at position two of SortedList is " + items.GetByIndex(1);
```

Listing 18.173 Using the SortedList.GetByIndex() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

MyLabel.Text = "Value at position two of SortedList is " _
+ items.GetByIndex(1).ToString()
```

SortedList.GetKey()

Syntax

```
Object GetKey( Int32 index)
```

Description

The `SortedList.GetKey()` method returns the key of the object at any location in the sortedlist that is given by the index argument.

Example

Listings 18.174 and 18.175 demonstrate how to use the `SortedList.GetKey()` method.

Listing 18.174 Using the SortedList.GetKey() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyLabel.Text = "Key at position two of SortedList is " + items.GetKey(1);
```

Listing 18.175 Using the SortedList.GetKey() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

MyLabel.Text = "Key at position two of SortedList is " + items.GetKey(1)
```

SortedList.GetKeyList()

Syntax

IList GetKeyList()

Description

The `SortedList.GetKeyList()` method returns a list of all keys in the sortedlist.

Example

Listings 18.176 and 18.177 demonstrate how to use the `SortedList.GetKeyList()` method.

Listing 18.176 Using the SortedList.GetKeyList() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

ArrayList MyArray = new ArrayList();
MyArray = ArrayList.Adapter(items.GetKeyList());
```

Listing 18.177 Using the SortedList.GetKeyList() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

dim MyArray as new ArrayList()
MyArray = ArrayList.Adapter(items.GetKeyList())
```

SortedList.GetValueList()

Syntax

IList GetValueList()

Description

The `SortedList.GetValueList()` method returns a list of all values in the sortedlist.

Example

Listings 18.178 and 18.179 demonstrate how to use the `SortedList.GetValueList()` method.

Listing 18.178 Using the SortedList.GetValueList() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

ArrayList MyArray = new ArrayList();
MyArray = ArrayList.Adapter(items.GetValueList());
```

Listing 18.179 Using the SortedList.GetValueList() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

dim MyArray as new ArrayList()
MyArray = ArrayList.Adapter(items.GetValueList())
```

SortedList.IndexOfKey()

Syntax

```
Int32 IndexOfKey( Object key )
```

Description

The `SortedList.IndexOfKey()` method returns the index of the object with the given key.

Example

Listings 18.180 and 18.181 demonstrate how to use the `SortedList.IndexOfKey()` method.

Listing 18.180 Using the SortedList.IndexOfKey() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyLabel.Text = "Index of 'key1' is '" + items.IndexOfKey("key1") + "'";
```

Listing 18.181 Using the SortedList.IndexOfKey() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
```

```
items.Add("key1", 4)

MyLabel.Text = "Index of 'key1' is '" +
    + items.IndexOfKey("key1").ToString() + "'"
```

SortedList.IndexOfValue()

Syntax

Int32 IndexOfValue(Object value)

Description

The `SortedList.IndexOfValue()` method returns the index of the object with the given value. It returns -1 if the value is not found.

Example

Listings 18.182 and 18.183 demonstrate how to use the `SortedList.IndexOfValue()` method.

Listing 18.182 Using the SortedList.IndexOfValue() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyLabel.Text = "Index of '1' is '" + items.IndexOfKey("1") + "'";
```

Listing 18.183 Using the SortedList.IndexOfValue() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

MyLabel.Text = "Index of '1' is '" + items.IndexOfValue("1").ToString() _ 
    + "'"
    -1 means not found
```

SortedList.Remove()

Syntax

Void Remove(Object key)

Description

The `SortedList.Remove()` method removes the object with the given key from the sortedlist.

Example

Listings 18.184 and 18.185 demonstrate how to use the `SortedList.Remove()` method.

Listing 18.184 Using the `SortedList.Remove()` Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

items.Remove("key1");
```

Listing 18.185 Using the `SortedList.Remove()` Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

items.Remove("key1")
```

SortedList.RemoveAt()

Syntax

```
Void RemoveAt( Int32 index )
```

Description

The `SortedList.RemoveAt()` method removes an object from the sortedlist at the given index. If the `SortedList.RemoveAt()` method is attempted on an empty sortedlist, `ArgumentOutOfRangeException` is thrown at runtime.

Example

Listings 18.186 and 18.187 demonstrate how to use the `SortedList.RemoveAt()` method.

Listing 18.186 Using the `SortedList.RemoveAt()` Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

items.RemoveAt(2);
```

Listing 18.187 Using the `SortedList.RemoveAt()` Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
```

Listing 18.187 continued

```
items.Add("key3", 1)
items.Add("key1", 4)

items.RemoveAt(2)
```

SortedList.SetByIndex()**Syntax**

Void SetByIndex(Int32 index, Object value)

Description

The `SortedList.SetByIndex()` method enables a developer to set the value of any index in a sortedlist by index.

Example

Listings 18.188 and 18.189 demonstrate how to use the `SortedList.SetByIndex()` method.

Listing 18.188 Using the SortedList.SetByIndex() Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

items.SetByIndex(1, "newvalue");
```

Listing 18.189 Using the SortedList.SetByIndex() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

items.SetByIndex(1, "newvalue")
```

SortedList.Synchronized()**Syntax**

SortedList Synchronized(SortedList list)

Description

The `SortedList.Synchronized()` method returns a synchronized copy of the current sortedlist.

Example

Listings 18.190 and 18.191 demonstrate how to use the `SortedList.Synchronized()` method.

Listing 18.190 Using the `SortedList.Synchronized()` Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

items = ArrayList.Synchronized(items);
MyLabel.Text = "True if items is synchronized: " + items.IsSynchronized;
```

Listing 18.191 Using the `SortedList.Synchronized()` Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

items = SortedList.Synchronized(items)
MyLabel.Text = "True if items is synchronized: " _
    + items.IsSynchronized.ToString()
```

SortedList.TrimToSize()

Syntax

```
Void TrimToSize()
```

Description

The `SortedList.TrimToSize()` method sets the capacity of the sortedlist to the current count of items in the sortedlist.

Example

Listings 18.192 and 18.193 demonstrate how to use the `SortedList.TrimToSize()` method.

Listing 18.192 Using the `SortedList.TrimToSize()` Method (C#)

```
SortedList items = new SortedList();
items.Add("key2", 6);
items.Add("key3", 1);
items.Add("key1", 4);

MyLabel.Text = "Current Capacity is " + items.Capacity.ToString();
items.TrimToSize();
MyLabel2.Text = "New Capacity is " + items.Capacity.ToString();
```

Listing 18.193 Using the SortedList.TrimToSize() Method (Visual Basic.NET)

```
dim items as new SortedList()
items.Add("key2", 6)
items.Add("key3", 1)
items.Add("key1", 4)

MyLabel.Text = "Current Capacity is " + items.Capacity.ToString()
items.TrimToSize()
MyLabel2.Text = "New Capacity is " + items.Capacity.ToString()
```

The Stack Class

The Stack class provides a number of properties and methods that you use to work with a *stack*, which is an abstract data structure that uses a first in, last out (FILO) structure. An object placed in a stack will be returned (that is, “popped”) only after all other objects that were placed in the stack, after the initial object, have been returned. The properties and methods of the Stack class are listed in Table 18.6.

Table 18.6 Properties and Methods of the Stack Class

Item	Description
Properties	
Count	Contains the number of items currently in the stack
IsReadOnly	Contains True if the stack is read-only and False if it is not
IsSynchronized	Contains True if the stack is synchronized
Methods	
Clear()	Clears all items from the stack
Clone()	Returns a duplicate of the current stack
Contains()	Returns True if the given value is found in the stack
CopyTo()	Copies all or part of the current stack to another stack that is passed in as an argument
Peek()	Returns the top object on the stack
Pop()	Returns the top object on the stack and then removes the object from the stack
Push()	Adds an object to the top of the stack
Synchronized()	Returns a synchronized copy of the stack
ToArray()	Copies all or part of the current stack to an array

Stack.Count

Syntax

Int32 Count

Description

The Stack.Count property contains the total number of objects in the stack.

Example

Listings 18.194 and 18.195 demonstrate how to use the Stack.Count property.

Listing 18.194 Using the stack.Count Property (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
myStack.Push("item3");

MyLabel.Text = "Count is " + myStack.Count.ToString();
```

Listing 18.195 Using the stack.Count Property (Visual Basic.NET)

```
dim myStack as new Stack()
myStack.Push("item1")
myStack.Push(4)
myStack.Push("item3")

MyLabel.Text = "Count is " + myStack.Count.ToString()
```

Stack.IsReadOnly

Syntax

```
Boolean IsReadOnly
```

Description

The Stack.IsReadOnly property returns True if the stack is read-only.

Example

Listings 18.196 and 18.197 demonstrate how to use the Stack.IsReadOnly property.

Listing 18.196 Using the stack.IsReadOnly Property (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
myStack.Push("item3");

MyLabel.Text = "IsReadOnly set to " + myStack.IsReadOnly;
```

Listing 18.197 Using the stack.IsReadOnly Property (Visual Basic.NET)

```
dim myStack as new Stack()
myStack.Push("item1")
myStack.Push(4)
myStack.Push("item3")

MyLabel.Text = "IsReadOnly set to " + myStack.IsReadOnly.ToString()
```

Stack.IsSynchronized

Syntax

```
Boolean IsSynchronized
```

Description

The Stack.IsSynchronized property returns True if the stack is synchronized.

Example

Listings 18.198 and 18.199 demonstrate how to use the Stack.IsSynchronized property.

Listing 18.198 Using the Stack.IsSynchronized Property (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
myStack.Push("item3");

MyLabel.Text = "IsSynchronized set to " + myStack.IsSynchronized;
```

Listing 18.199 Using the Stack.IsSynchronized Property (Visual Basic.NET)

```
dim myStack as new Stack()
myStack.Push("item1")
myStack.Push(4)
myStack.Push("item3")

MyLabel.Text = "IsSynchronized set to " + myStack.IsSynchronized.ToString()
```

Stack.Clear()

Syntax

```
Void Clear()
```

Description

The Stack.Clear() method removes all objects from the stack.

Example

Listings 18.200 and 18.201 demonstrate how to use the Stack.Clear() method.

Listing 18.200 Using the Stack.Clear() Method (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
```

Listing 18.200 continued

```
myStack.Push("item3");  
  
myStack.Clear();
```

Listing 18.201 Using the stack.Clear() Method (Visual Basic.NET)

```
dim myStack as new Stack()  
myStack.Push("item1")  
myStack.Push(4)  
myStack.Push("item3")  
  
myStack.Clear()
```

Stack.Clone()

Syntax

Object Clone()

Description

The Stack.Clone() method returns a copy of the current stack.

Example

Listings 18.202 and 18.203 demonstrate how to use the Stack.Clone() method.

Listing 18.202 Using the stack.Clone() Method (C#)

```
Stack myStack = new Stack();  
myStack.Push("item1");  
myStack.Push(4);  
myStack.Push("item3");  
  
Stack newStack = (Stack) myStack.Clone();
```

Listing 18.203 Using the stack.Clone() Method (Visual Basic.NET)

```
dim myStack as new Stack()  
myStack.Push("item1")  
myStack.Push(4)  
myStack.Push("item3")  
  
Dim newStack as new Stack()  
newStack = myStack.Clone()
```

Stack.Contains()

Syntax

Boolean Contains(*Object* obj)

Description

The Stack.Contains() method returns True if the obj argument is found in the stack.

Example

Listings 18.204 and 18.205 demonstrate how to use the Stack.Contains() method.

Listing 18.204 Using the Stack.Contains() Method (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
myStack.Push("item3");

MyLabel.Text = "Does list contain this object: "
    + myStack.Contains(4).ToString();
```

Listing 18.205 Using the Stack.Contains() Method (Visual Basic.NET)

```
dim myStack as new Stack()
myStack.Push("item1")
myStack.Push(4)
myStack.Push("item3")

MyLabel.Text = "Does list contain this object: " _
    + myStack.Contains(4).ToString()
```

Stack.CopyTo()

Syntax

```
Void CopyTo( Array array, Int32 index )
```

Description

The Stack.CopyTo() method copies the stack to an array.

Example

Listings 18.206 and 18.207 demonstrate how to use the Stack.CopyTo() method.

Listing 18.206 Using the Stack.CopyTo() Method (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
myStack.Push("item3");

String[] MyArray = new String[4];
items.CopyTo(MyArray);
MyLabel.Text = "Third item of ArrayList is " + MyArray[2];
```

Listing 18.207 Using the Stack.CopyTo() Method (Visual Basic.NET)

```
dim myStack as new Stack()
myStack.Push("item1")
myStack.Push(4)
myStack.Push("item3")

Dim MyArray(4) as String
myStack.CopyTo(MyArray, 0)
MyLabel.Text = "Third item of ArrayList is " + MyArray(2)
```

Stack.Peek()

Syntax

Object Peek()

Description

The Stack.Peek() method returns the top object in the stack, but it does not remove this item from the stack.

Example

Listings 18.208 and 18.209 demonstrate how to use the Stack.Peek() method.

Listing 18.208 Using the Stack.Peek() Method (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
myStack.Push("item3");

MyLabel.Text = "Top object on the stack is '" + myStack.Peek() + "'";
```

Listing 18.209 Using the Stack.Peek() Method (Visual Basic.NET)

```
dim myStack as new Stack()
myStack.Push("item1")
myStack.Push(4)
myStack.Push("item3")

MyLabel.Text = "Top object on the stack is '" + myStack.Peek() + "'"
```

Stack.Pop()

Syntax

Object Pop()

Description

The Stack.Pop() method returns the top item on the stack and then removes it from the stack.

Example

Listings 18.210 and 18.211 demonstrate how to use the `Stack.Pop()` method.

Listing 18.210 Using the Stack.Pop() Method (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
myStack.Push("item3");

myStack.Pop();
```

Listing 18.211 Using the Stack.Pop() Method (Visual Basic.NET)

```
dim myStack as new Stack()
myStack.Push("item1")
myStack.Push(4)
myStack.Push("item3")

myStack.Pop()
```

Stack.Push()

Syntax

```
Void Push( Object obj )
```

Description

The `Stack.Push()` method places an item on top of the stack.

Example

Listings 18.212 and 18.213 demonstrate how to use the `Stack.Push()` method.

Listing 18.212 Using the Stack.Push() Method (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
myStack.Push("item3");

myStack.Push("newItem");
myStack.Push(9);
```

Listing 18.213 Using the Stack.Push() Method (Visual Basic.NET)

```
dim myStack as new Stack()
myStack.Push("item1")
myStack.Push(4)
myStack.Push("item3")
```

Listing 18.213 continued

```
myStack.Push("newItem")
myStack.Push(9)
```

Stack.Synchronized()

Syntax

```
Stack Synchronized( Stack stack )
```

Description

The `Stack.Synchronized()` method returns a synchronized version of the stack.

Example

Listings 18.214 and 18.215 demonstrate how to use the `Stack.Synchronized()` method.

Listing 18.214 Using the Stack.Synchronized() Method (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
myStack.Push("item3");

myStack = Stack.Synchronized(myStack);
MyLabel.Text = "True if myStack is synchronized: " + myStack.IsSynchronized;
```

Listing 18.215 Using the Stack.Synchronized() Method (Visual Basic.NET)

```
dim myStack as new Stack()
myStack.Push("item1")
myStack.Push(4)
myStack.Push("item3")

myStack = Stack.Synchronized(myStack)
MyLabel.Text = "True if myStack is synchronized: " _
    + myStack.IsSynchronized.ToString()
```

Stack.ToArray()

Syntax

```
Object[] ToArray()
```

Description

The `Stack.ToArray()` method returns the contents of the stack in an array of objects.

Example

Listings 18.216 and 18.217 demonstrate how to use the `Stack.ToArray()` method.

Listing 18.216 Using the Stack.ToArray() Method (C#)

```
Stack myStack = new Stack();
myStack.Push("item1");
myStack.Push(4);
myStack.Push("item3");

Object[] MyArray;
MyArray = myStack.ToArray();
```

Listing 18.217 Using the Stack.ToArray() Method (Visual Basic.NET)

```
dim myStack as new Stack()
myStack.Push("item1")
myStack.Push(4)
myStack.Push("item3")

dim MyArray() as Object
MyArray = myStack.ToArray()
```


CHAPTER 19

System.Data.SqlClient Reference

The `System.Data.SqlClient` namespace contains all the objects and methods necessary to perform operations on a SQL database. The `SqlConnection` class handles connecting to a Microsoft SQL database. The `SqlCommand` class is used to execute SQL commands against a database. `SqlDataReader` is a powerful class used to read large amounts of data from a SQL database. The `SqlTransaction` class enables you to easily perform SQL transactions.

Because the `System.Data.SqlClient` namespace uses a managed provider to connect to a SQL server, it can only be used to connect to Microsoft SQL Server 7.0 and greater. To connect to an Oracle database or any other OLE DB source, you use the `System.Data.OleDb` namespace.

The examples in this chapter do not stand alone. They work only when placed into the `Page_Load()` events in Listings 19.1 and 19.2, for C# and Visual Basic.NET, respectively.

Listing 19.1 Stub Code for System.Data.SqlClient C# Examples

```
<% @Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css"
        href="Main.css">
    <!-- End Style Sheet -->
```

Listing 19.1 continued

```
<script language="C#" runat="server" >
    void Page_Load(Object Source, EventArgs E)
    {

        //Place System.Data.SqlClient C# Code here

    }
</script>

</HEAD>
<BODY>

<h1>System.Data.SqlClient Examples</h1>
<hr>

<form runat="server" id=form1 name=form1>
    <asp:Label id=msg runat="server"></asp:Label>
</form>
<hr>

</BODY>
</HTML>
```

Listing 19.2 Stub Code for System.Data.SqlClient Visual Basic.NET Examples

```
<% @Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >
        Sub Page_Load(Source as Object, E as EventArgs)

            'Place System.Data.SqlClient Visual Basic.NET Code here

        End Sub
    </script>

</HEAD>
<BODY>

<h1>System.Data.SqlClient Examples</h1>
```

Listing 19.2 continued

```
<hr>

<form runat="server" id=form1 name=form1>
    <asp:Label id=msg runat="server"></asp:Label>
</form>
<hr>

</BODY>
</HTML>
```

The SqlCommand Class

The `SqlCommand` class is at the heart of the `System.SqlClient` namespace. It is used to execute operations on a database and retrieve data (see Table 19.1).

Table 19.1 Properties and Methods of the `SqlCommand` Class

Item	Description
Properties	
<code>CommandText</code>	Contains the text of a SQL query
<code>CommandTimeout</code>	Contains the length of the timeout of a query, in seconds
<code> CommandType</code>	Specifies the type of command to be executed
<code> Connection</code>	Specifies the connection to the database
<code> Parameters</code>	Specifies a collection of parameters for the SQL query
<code> Transaction</code>	Specifies a transaction object, which enables developers to run queries in a transaction
Methods	
<code>Cancel()</code>	Cancels the running query
<code>CreateParameter()</code>	Returns a new SQL parameter
<code>ExecuteNonQuery()</code>	Executes the <code>CommandText</code> property against the database and does not return a result set
<code>ExecuteReader()</code>	Executes the <code>CommandText</code> property and returns data in a <code>DataReader</code> object
<code>ExecuteScalar()</code>	Executes the <code>CommandText</code> property and returns a single value
<code>ExecuteXmlReader()</code>	Executes the <code>CommandText</code> property and returns data in an <code>XMLDataReader</code> object
<code>ResetCommandTimeout()</code>	Resets the <code>CommandTimeout</code> property for the query

`SqlCommand.CommandText`

Syntax

`String CommandText`

Description

The CommandText property is a string that contains the text of a database query.

Example

Listings 19.3 and 19.4 demonstrate how to set the CommandText property of a SqlCommand object.

Listing 19.3 Using the CommandText Property (C#)

```
SqlCommand nwCmd = new SqlCommand();
nwCmd.CommandText = "DELETE FROM Employees";
```

Listing 19.4 Using the CommandText Property (Visual Basic.NET)

```
dim nwCmd as new SqlCommand()
nwCmd.CommandText = "DELETE FROM Employees"
```

SqlCommand.CommandTimeout

Syntax

Int32 CommandTimeout

Description

The CommandTimeout property contains the number of seconds before the query will timeout. By default, CommandTimeout is set to 30 seconds.

Example

Listings 19.5 and 19.6 demonstrate how to set the CommandTimeout property.

Listing 19.5 Setting the CommandTimeout Property (C#)

```
SqlCommand nwCmd = new SqlCommand();
nwCmd.CommandTimeout = 40;
```

Listing 19.6 Setting the CommandTimeout Property (Visual Basic.NET)

```
dim nwCmd as new SqlCommand()
nwCmd.CommandTimeout = 40
```

SqlCommand.CommandType

Syntax

CommandType CommandType

Description

The CommandType property contains the type of query being performed. CommandType is defined in the System.Data namespace. The CommandType property can be set to StoredProcedure, TableDirect, or Text.

Example

Listings 19.7 and 19.8 demonstrate how to set the CommandType property of a SqlCommand object.

Listing 19.7 Setting the CommandType Property (C#)

```
SqlCommand nwCmd = new SqlCommand();
nwCmd.CommandType = CommandType.Text;
```

Listing 19.8 Setting the CommandType Property (Visual Basic.NET)

```
dim nwCmd as new SqlCommand()
nwCmd.CommandType = CommandType.Text
```

SqlCommand.Connection

Syntax

```
SqlConnection Connection
```

Description

The Connection property is set to a valid SqlConnection object against which the query is to be performed. The easiest way to set the connection for the SqlCommand object is to explicitly create a SqlConnection object and assign it to the Connection property, as shown in Listings 19.7 and 19.8.

Example

Listings 19.9 and 19.10 demonstrate how to explicitly create a SqlConnection object and assign it to the Connection property of a SqlCommand object.

Listing 19.9 Creating a Connection (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand();
nwCmd.Connection = nwConn;
```

Listing 19.10 Creating a Connection (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; User Id=sa; _
    Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand()
nwCmd.Connection = nwConn
```

SqlCommand.Parameters

Syntax

```
SqlParameterCollection Parameters
```

Description

The `Parameters` property contains the names of a group of SQL parameters that are used when calling a stored procedure that requires input or output parameters.

Example

Listings 19.11 and 19.12 demonstrate how to use the `Parameters` property to build the parameters needed to call a stored procedure.

Listing 19.11 Using the Parameters Property (C#)

```
//Create and Open Connection
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    UID=sa;PWD=;Initial Catalog=northwind;"); nwConn.Open();

//Create Command
SqlCommand nwCmd = new SqlCommand("AddNewEmployee", nwConn);
nwCmd.CommandType = CommandType.StoredProcedure;

//Configure input parameters
SqlParameter nwParam = new SqlParameter();
nwParam = nwCmd.Parameters.Add(new SqlParameter("@LastName",
    SqlDbType.NVarChar, 20));
nwParam.Direction = ParameterDirection.Input;
nwParam.Value = sLastName.Text;

//Configure output parameters
nwParam = nwCmd.Parameters.Add(new SqlParameter("@retval",
    SqlDbType.Int, 4)); nwParam.Direction = ParameterDirection.Output;

nwCmd.ExecuteNonQuery();
```

Listing 19.12 Using the Parameters Property (Visual Basic.NET)

```
'Create and Open Connection
dim nwConn as new SqlConnection("Data Source=localhost; _
    UID=sa;PWD=;Initial Catalog=northwind;")
nwConn.Open()

'Create Command
dim nwCmd as new SqlCommand("AddNewEmployee", nwConn)
nwCmd.CommandType = CommandType.StoredProcedure

'Configure input parameters
dim nwParam as SqlParameter()
```

Listing 19.12 continued

```

nwParam = nwCmd.Parameters.Add(new SqlParameter("@LastName", _
    SqlDbType.NVarChar, 20))
nwParam.Direction = ParameterDirection.Input
nwParam.Value = sLastName.Text

'Configure output parameters
nwParam = nwCmd.Parameters.Add(new SqlParameter("@retval", SqlDbType.Int, 4))
nwParam.Direction = ParameterDirection.Output

nwCmd.ExecuteNonQuery()

```

SqlCommand.Transaction

Syntax

SqlTransaction Transaction

Description

The *Transaction* property contains the transaction object (if any) for the current command set.

Example

Listings 19.13 and 19.14 demonstrate how to set the *Transaction* property for a command object.

Listing 19.13 Setting the Transaction Property (C#)

```

SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlTransaction nwTrans = nwConn.BeginTransaction(IsolationLevel.Serializable,
    "RemoveEmployees");

SqlCommand nwCmd = new SqlCommand("", nwConn);
nwCmd.Transaction = nwTrans;

```

Listing 19.14 Setting the Transaction Property (Visual Basic.NET)

```

SqlConnection nwConn = new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwTrans as object
nwTrans = nwConn.BeginTransaction(IsolationLevel.Serializable, _
    "RemoveEmployees")

dim nwCmd as new SqlCommand("", nwConn)
nwCmd.Transaction = nwTrans

```

SqlCommand.Cancel()

Syntax

```
Void Cancel()
```

Description

The `Cancel()` method cancels a running query.

Example

Listings 19.15 and 19.16 demonstrate canceling running queries through C# and Visual Basic.NET.

Listing 19.15 Canceling a Running Query (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT * FROM ReallyBigTable", nwConn);
nwCmd.ExecuteNonQuery();
nwCmd.Cancel();
```

Listing 19.16 Canceling a Running Query (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT * FROM ReallyBigTable", nwConn)
nwCmd.ExecuteNonQuery()
nwCmd.Cancel()
```

SqlCommand.CreateParameter()

Syntax

```
SqlParameter CreateParameter()
```

Description

The `CreateParameter()` method returns a new instance of a SQL parameter.

Example

Listings 19.17 and 19.18 demonstrate how to use the `CreateParameter()` method to help build a set of parameters needed to call a stored procedure.

Listing 19.17 Building a SQL Parameter, Using the CreateParameter() Method (C#)

```
SqlParameter param = nwCmd.CreateParameter();
param.ParameterName = "@CustomerID";
param.DbType = SqlDbType.NVarChar;
param.Size = 5;
param.Value = "12345";
param.Direction = ParameterDirection.Input;

nwCmd.Parameters.Add(param);
```

Listing 19.18 Building a SQL Parameter, Using the CreateParameter() Method (Visual Basic.NET)

```
dim param as new SqlParameter()
param = nwCmd.CreateParameter()
param.ParameterName = "@CustomerID"
param.DbType = SqlDbType.NVarChar
param.Size = 5
param.Value = "12345"
param.Direction = ParameterDirection.Input

nwCmd.Parameters.Add(param)
```

SqlCommand.ExecuteNonQuery()

Syntax

Int32 ExecuteNonQuery()

Description

The `ExecuteNonQuery()` method executes the command text against the database specified in the `Connection` object. This method is optimized for queries that do not return any information (for example, `DELETE` and `UPDATE` queries).

Example

Listings 19.19 and 19.20 demonstrate how to use `ExecuteNonQuery()` to perform a `DELETE` query on a table in the Northwind database.

Listing 19.19 Using the ExecuteNonQuery() Method (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("DELETE FROM Employees WHERE
    LastName='Tomb'", nwConn); nwCmd.ExecuteNonQuery();
```

Listing 19.20 Using the ExecuteNonQuery() Method (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; User Id=sa; _
    Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("DELETE FROM Employees WHERE LastName='Tomb'", _
    nwConn) nwCmd.ExecuteNonQuery()
```

SqlCommand.ExecuteReader()

Syntax

```
SqlDataReader ExecuteReader()
SqlDataReader ExecuteReader( CommandBehavior behavior )
```

Description

The `ExecuteReader()` method executes the command text against the database specified in the `Connection` object and returns a `SqlDataReader` object with the results of the query.

Example

Listings 19.21 and 19.22 demonstrate how to use the `ExecuteReader()` method to return a `SqlDataReader` object.

Listing 19.21 Using the ExecuteReader() Method (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT EmployeeID,
    LastName, FirstName FROM Employees", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();
```

Listing 19.22 Using the ExecuteReader() Method (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID, LastName, _
    FirstName FROM Employees", nwConn)

SqlDataReader nwReader = nwCmd.ExecuteReader()
```

SqlCommand.ExecuteScalar()

Syntax

```
Object ExecuteScalar()
```

Description

The `ExecuteScalar()` method executes the command text against the database specified in the `Connection` object and returns a single object. The `ExecuteScalar()` method exists because it is wasteful to return a dataset for a single value. The overhead for the dataset would be much larger than the actual value being returned.

Example

Listings 19.23 and 19.24 demonstrate how to use the `ExecuteScalar()` method to retrieve the number of records in a table in the Northwind database.

Listing 19.23 Using the ExecuteScalar() Method (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT count(*) FROM Employees", nwConn);
Int32 employeeCount = (Int32)nwCmd.ExecuteScalar();

msg.Text = "There are " + employeeCount.ToString()
    + " employees in the database.";
```

Listing 19.24 Using the ExecuteScalar() Method (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT count(*) FROM Employees", nwConn)
dim employeeCount as int
employeeCount = nwCmd.ExecuteScalar()

msg.Text = "There are " + employeeCount.ToString() _
    + " employees in the database."
```

SqlCommand.ExecuteNonQuery()

Syntax

```
XmlReader ExecuteXmlReader()
```

Description

The `ExecuteXmlReader()` method executes the command text against the database specified in the `Connection` object and returns the result, set in an `XmlReader` object.

Example

Listings 19.25 and 19.26 demonstrate how to use the `ExecuteXmlReader()` method to retrieve a result set.

Listing 19.25 Using the ExecuteXmlReader() Method (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT EmployeeID,
    LastName, FirstName FROM Employees", nwConn);
XmlReader nwReader = nwCmd.ExecuteXmlReader();
```

Listing 19.26 Using the ExecuteXmlReader() Method (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; User Id=sa; _
    Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID, LastName, _
    FirstName FROM Employees", nwConn)
dim nwReader as nwCmd.ExecuteXmlReader()
```

SqlCommand.ResetCommandTimeout()

Syntax

```
Void ResetCommandTimeout()
```

Description

The `ResetCommandTimeout()` method resets the `CommandTimeout` property to the default value of 30 seconds.

Example

Listings 19.27 and 19.28 demonstrate how to use the `ResetCommandTimeout()` method.

Listing 19.27 Using the ResetCommandTimeout() Method (C#)

```
SqlCommand nwCmd = new SqlCommand("DELETE FROM Employees
    WHERE LastName='Tomb'", nwConn);
nwCmd.CommandTimeout = 45;
nwCmd.ResetCommandTimeout();
//CommandTimeout is now reset to 30
```

Listing 19.28 Using the ResetCommandTimeout() Method (Visual Basic.NET)

```
dim nwCmd as new SqlCommand("DELETE FROM Employees _
    WHERE LastName='Tomb'", nwConn);
nwCmd.CommandTimeout = 45
nwCmd.ResetCommandTimeout()
'CommandTimeout is now reset to 30
```

The SqlConnection Class

The SqlConnection class provides all the properties and methods needed to make a connection to a Microsoft SQL database. Table 19.2 shows the properties and methods in the SqlConnection class and briefly describes the purpose of each.

Table 19.2 Properties and Methods of the SqlConnection Class

Item	Description
Properties	
ConnectionString	Contains the database connection string
ConnectionTimeout	Contains the timeout for the connection in seconds
Database	Contains the name of the database to connect to
DataSource	Contains the name of the connected server
PacketSize	Contains the size of packets used to communicate with the server
ServerVersion	Contains the version of SQL that the server is running
State	Contains the state of the connection
WorkstationId	Contains the NetBIOS identifier of the machine hosting the Web form
Methods	
BeginTransaction()	Places the connection into a transaction and returns the newly created SqlTransaction object
ChangeDatabase()	Enables the developer to change to a different database programmatically
Close()	Closes the connection to the current database
CreateCommand()	Returns a new command object
Open()	Opens the connection to the database

SqlConnection.ConnectionString

Syntax

```
String ConnectionString
```

Description

The ConnectionString property contains the database connection string.

Example

Listings 19.29 and 19.30 demonstrate how to specify the ConnectionString property.

Listing 19.29 Specifying the ConnectionString Property (C#)

```
SqlConnection nwConn = new SqlConnection();
nwConn.ConnectionString = "Data Source=localhost; User Id=sa;
    Password=;Initial Catalog=northwind";
```

Listing 19.30 Specifying the ConnectionString Property (Visual Basic.NET)

```
dim nwConn as new SqlConnection()  
nwConn.ConnectionString = "Data Source=localhost; User Id=sa; _  
    Password=;Initial Catalog=northwind"
```

SqlConnection.ConnectionTimeout

Syntax

Int32 ConnectionTimeout

Description

`ConnectionTimeout` is a read-only property that contains the length of time in seconds before the connection times out. If you need to modify the length of time before the connection times out, change the `Connection Timeout` setting in your connection string. The default for this property is 30 seconds.

Example

Listings 19.31 and 19.32 demonstrate how to retrieve the `ConnectionTimeout` property.

Listing 19.31 Retrieving the ConnectionTimeout Property Value (C#)

```
SqlConnection nwConn = new SqlConnection();  
msg.Text = nwConn.ConnectionTimeout;
```

Listing 19.32 Retrieving the ConnectionTimeout Property Value (Visual Basic.NET)

```
dim nwConn = new SqlConnection()  
msg.Text = nwConn.ConnectionTimeout
```

SqlConnection.Database

Syntax

String Database

Description

`Database` is a read-only property that contains the name of the connected database. To change databases, you need to use the `ChangeDatabase()` method or change the connection string.

Example

Listings 19.33 and 19.34 demonstrate how to programmatically retrieve the `Database` value.

Listing 19.33 Retrieving the Database Property Value (C#)

```
SqlConnection nwConn = new SqlConnection();  
msg.Text = nwConn.Database;
```

Listing 19.34 Retrieving the Database Property Value (Visual Basic.NET)

```
dim nwConn as new SqlConnection()  
msg.Text = nwConn.Database
```

SqlConnection.DataSource

Syntax

String DataSource

Description

DataSource is a read-only property that contains the name of the data source you are connecting to. To change the data source, you need to modify your connection string.

Example

Listings 19.35 and 19.36 demonstrate how to programmatically retrieve the DataSource property value.

Listing 19.35 Retrieving the DataSource Property Value (C#)

```
SqlConnection nwConn = new SqlConnection();  
msg.Text = nwConn.DataSource;
```

Listing 19.36 Retrieving the DataSource Property Value (Visual Basic.NET)

```
dim nwConn as new SqlConnection()  
msg.Text = nwConn.DataSource
```

SqlConnection.PacketSize

Syntax

Int32 PacketSize

Description

PacketSize is a read-only property that contains the size of packets used in communication with the server.

Example

Listings 19.37 and 19.38 demonstrate how to programmatically retrieve the PacketSize property value.

Listing 19.37 Retrieving the PacketSize Property Value (C#)

```
SqlConnection nwConn = new SqlConnection();  
msg.Text = nwConn.PacketSize;
```

Listing 19.38 Retrieving the PacketSize Property Value (Visual Basic.NET)

```
dim nwConn as new SqlConnection()  
msg.Text = nwConn.PacketSize
```

SqlConnection.ServerVersion

Syntax

String ServerVersion

Description

`ServerVersion` is a read-only property that contains the version number of SQL server that the server is running.

Example

Listings 19.39 and 19.40 demonstrate how to programmatically retrieve the `ServerVersion` property value.

Listing 19.39 Retrieving the ServerVersion Property Value (C#)

```
SqlConnection nwConn = new SqlConnection();  
msg.Text = nwConn.ServerVersion;
```

Listing 19.40 Retrieving the ServerVersion Property Value (Visual Basic.NET)

```
dim nwConn as new SqlConnection()  
msg.Text = nwConn.ServerVersion
```

SqlConnection.State

Syntax

ConnectionState State

Description

The `State` property contains detailed information about the current state of the connection. It can contain the following values: `Broken`, `Closed`, `Connecting`, `Executing`, `Fetching`, and `Open`. These values can be found in the `System.Data.ConnectionState` namespace.

Example

Listings 19.41 and 19.42 demonstrate how to retrieve the value of the `State` property.

Listing 19.41 Retrieving the Value of the state Property (C#)

```
SqlConnection nwConn = new SqlConnection();
msg.Text = nwConn.State.ToString();
```

Listing 19.42 Retrieving the Value of the state Property (Visual Basic.NET)

```
dim nwConn as new SqlConnection()
msg.Text = nwConn.State.ToString()
```

SqlConnection.WorkstationId

Syntax

String WorkstationId

Description

The WorkstationId property contains the NetBIOS name of the computer originating the SQL connection.

Example

Listings 19.43 and 19.44 demonstrate how to retrieve the value of the WorkstationId property.

Listing 19.43 Retrieving the Value of the workstationId Property (C#)

```
SqlConnection nwConn = new SqlConnection();
msg.Text = nwConn.WorkstationId;
```

Listing 19.44 Retrieving the Value of the workstationId Property (Visual Basic.NET)

```
dim nwConn as new SqlConnection()
msg.Text = nwConn.WorkstationId
```

SqlConnection.BeginTransaction()

Syntax

```
SqlTransaction BeginTransaction( IsolationLevel iso )
SqlTransaction BeginTransaction()
SqlTransaction BeginTransaction( IsolationLevel iso, String transactionName )
SqlTransaction BeginTransaction( String transactionName )
```

Description

The BeginTransaction() method places the current connection under a transaction.

Example

Listings 19.45 and 19.46 demonstrate how to use the BeginTransaction() method to initiate a transaction.

Listing 19.45 Beginning a Transaction (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
SqlTransaction nwTrans = nwConn.BeginTransaction(IsolationLevel.Serializable,  
"RemoveEmployees");
```

Listing 19.46 Beginning a Transaction (Visual Basic.NET)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost; _  
User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwTrans as object  
nwTrans = nwConn.BeginTransaction(IsolationLevel.Serializable, _  
"RemoveEmployees")
```

SqlConnection.ChangeDatabase()

Syntax

```
Boolean ChangeDatabase( String database )
```

Description

The `ChangeDatabase()` method enables a developer to change databases programmatically, without having to create a new connection object.

Example

Listings 19.47 and 19.48 demonstrate how to change databases programmatically.

Listing 19.47 Changing Databases Programmatically (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
nwConn.ChangeDatabase("pubs"); //Database now changed to pubs
```

Listing 19.48 Changing Databases Programmatically (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
nwConn.ChangeDatabase("pubs") //Database now changed to pubs
```

SqlConnection.Close()

Syntax

```
Void Close()
```

Description

The Close() method closes the connection to the database.

Example

Listings 19.49 and 19.50 demonstrate how to close the connection to the database.

Listing 19.49 Closing a Database Connection (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

nwConn.Close();
```

Listing 19.50 Closing a Database Connection (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

nwConn.Close()
```

SqlConnection.CreateCommand()

Syntax

```
SqlCommand CreateCommand()
```

Description

The CreateCommand() method returns a new instance of a SqlCommand object.

Example

Listings 19.51 and 19.52 demonstrate how to use the CreateCommand() method to create a new command object.

Listing 19.51 Creating a New Command Object (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = nwConn.CreateCommand();
nwCmd.CommandText = "DELETE FROM Employees";
```

Listing 19.52 Creating a New Command Object (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
    User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwCmd as new SqlCommand()  
nwCmd = nwConn.CreateCommand()  
nwCmd.CommandText = "DELETE FROM Employees"
```

SqlConnection.Open()

Syntax

Void Open()

Description

The Open() method opens the connection to the database by using the information provided in the ConnectionString property.

Example

Listings 19.53 and 19.54 demonstrate how to open a connection to a database.

Listing 19.53 Opening a Connection to a Database (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost; _  
    User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();
```

Listing 19.54 Opening a Connection to a Database (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
    User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()
```

The SqlDataReader Class

The SqlDataReader class defines a lightweight yet powerful object that is used to read information from a SQL database. A SqlDataReader object has a small footprint, because it doesn't contain more than a single record in memory at any time. This makes it ideal for reading large amounts of data from a database.

Table 19.3 Properties and Methods of the SqlDataReader Class

Type	Item Description
Properties	
FieldCount	Contains the number of fields retrieved from the query
IsClosed	Contains True if the SqlDataReader object is closed
Item	Contains A collection of values that are accessible both by field name and by ordinal number
RecordsAffected	Returns the number of records affected by an executed query

Table 19.3 continued

Type	Item Description
Methods	
Close()	Closes the SqlDataReader object
GetBoolean()	Retrieves a value of type Boolean
GetByte()	Retrieves a value of type Byte
GetBytes()	Retrieves values of type Byte
GetChar()	Retrieves a value of type Char
GetChars()	Retrieves values of type char
GetDataTypeName()	Retrieves the data type of a field by ordinal number
GetDateTime()	Retrieves a value of type DateTime
GetDecimal()	Retrieves a value of type Decimal
GetDouble()	Retrieves a value of type Double
GetFieldType()	Retrieves the .NET data type of a field by ordinal number
GetFloat()	Retrieves a value of type Float
GetGuid()	Retrieves a value of type GUID
GetInt16()	Retrieves a value of type Int
GetInt32()	Retrieves a value of type Int
GetInt64()	Retrieves a value of type Int
GetName()	Returns a field name by ordinal number
GetOrdinal()	Returns an ordinal number by field name
GetSchemaTable()	Returns a data table that contains a database schema
GetSqlBinary()	Retrieves a value of type SqlBinary
GetSqlBit()	Retrieves a value of type SqlBit
GetSqlByte()	Retrieves a value of type SqlByte
GetSqlDateTime()	Retrieves a value of type SqlDateTime
GetSqlDecimal()	Retrieves a value of type SqlDecimal
GetSqlDouble()	Retrieves a value of type Double
GetSqlGuid()	Retrieves a value of type SqlGuid
GetSqlInt16()	Retrieves a value of type SqlInt16
GetSqlInt32()	Retrieves a value of type SqlInt32
GetSqlInt64()	Retrieves a value of type SqlInt64
GetSqlMoney()	Retrieves a value of type SqlMoney
GetSqlSingle()	Retrieves a value of type SqlSingle
GetSqlString()	Retrieves a value of type SqlString
GetSqlValue()	Returns a SQL field value by ordinal number
GetString()	Retrieves a value of type String
GetValue()	Returns the value of field data by ordinal number
IsDBNull()	Returns True if the SQL field contains Null
NextResult()	Reads the next result in the result set into memory when reading batch T-SQL results
Read()	Reads the next result in the result set into memory

SqlDataReader.FieldCount

Syntax

```
Int32 FieldCount
```

Description

The `FieldCount` property contains the number of fields in the current record.

Example

Listings 19.55 and 19.56 demonstrate how to retrieve the number of fields in the current record.

Listing 19.55 Using the FieldCount Property (C#)

```
string message = "";
int FieldCount;

//Instantiate and open connection object
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

//Get command object
SqlCommand nwCmd = new SqlCommand("SELECT EmployeeID, LastName,
    FirstName FROM Employees", nwConn);

//Execute reader
SqlDataReader nwReader = nwCmd.ExecuteReader();

//Get FieldCount
FieldCount = nwReader.FieldCount;
```

Listing 19.56 Using the FieldCount Property (Visual Basic.NET)

```
dim message as string
dim FieldCount as Int32

'Instantiate and open connection object
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

'Get command object
dim nwCmd as new SqlCommand("SELECT EmployeeID, LastName, _
    FirstName FROM Employees", nwConn)

'Execute reader
dim nwReader as object
```

Listing 19.56 continued

```
nwReader = nwCmd.ExecuteReader()

'Get FieldCount
FieldCount = nwReader.FieldCount
```

SqlDataReader.IsClosed

Syntax

```
Boolean IsClosed
```

Description

The `IsClosed` property contains `True` if the `SqlDataReader` object is closed.

Example

Listings 19.57 and 19.58 demonstrate how to use the `IsClosed` property.

Listing 19.57 Using the IsClosed Property (C#)

```
string message = ""; int FieldCount;

//Instantiate and open connection object
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

//Get command object
SqlCommand nwCmd = new SqlCommand("SELECT EmployeeID, LastName,
    FirstName FROM Employees", nwConn);

//Execute reader
SqlDataReader nwReader = nwCmd.ExecuteReader();
if (nwReader.IsClosed != true) {
    //work with reader
}
```

Listing 19.58 Using the IsClosed Property (Visual Basic.NET)

```
dim message as string
dim FieldCount as Int32

'Instantiate and open connection object
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

'Get command object
dim nwCmd as new SqlCommand("SELECT EmployeeID, LastName, _
```

Listing 19.58 continued

```
FirstName FROM Employees", nwConn)

'Execute reader
dim nwReader as object
nwReader = nwCmd.ExecuteReader()
if nwReader.IsClosed <> true
    'work with reader
end if
```

SqlDataReader.Item

Syntax

```
Object Item( String name )
Object Item( Int32 i )
```

Description

The **Item** property retrieves the value of a column in its native data format.

Example

Listings 19.59 and 19.60 demonstrate how to use the item collection to access specific columns in the results of a **SqlDataReader** object.

Listing 19.59 Retrieving Column Values with the Item Property (C#)

```
string message = "";

//Instantiate and open connection object
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

//Get command object
SqlCommand nwCmd = new SqlCommand("SELECT EmployeeID, LastName,
    FirstName FROM Employees", nwConn);

//Execute reader
SqlDataReader nwReader = nwCmd.ExecuteReader();

//Loop through all records building a string
while (nwReader.Read()) {
    message = message + nwReader["EmployeeID"] + " " +
        nwReader["LastName"] + ", " + nwReader["FirstName"] + "<BR>";
}

nwReader.Close();
```

Listing 19.60 Retrieving Column Values with the Item Property (Visual Basic.NET)

```

dim message as string

'Instantiate and open connection object
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

'Get command object
dim nwCmd as new SqlCommand("SELECT EmployeeID, LastName, _
    FirstName FROM Employees", nwConn)

'Execute reader
dim nwReader as object
nwReader = nwCmd.ExecuteReader()

'Loop through all records building a string
while nwReader.Read()
    message = message + nwReader("EmployeeID") + " " +
        nwReader("LastName") + ", " + nwReader("FirstName") + "<BR>"
end while

nwReader.Close()

```

SqlDataReader.RecordsAffected

Syntax

Int32 RecordsAffected

Description

The RecordsAffected property contains the number of records affected by the query.

Example

Listings 19.61 and 19.62 demonstrate how to access the RecordsAffected property.

Listing 19.61 Using the RecordsAffected Property (C#)

```

int RecordsAffected;

//Instantiate and open connection object
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

```

Listing 19.61 continued

```
//Get command object
SqlCommand nwCmd = new SqlCommand("SELECT EmployeeID, LastName,
    FirstName FROM Employees", nwConn);

//Execute reader
SqlDataReader nwReader = nwCmd.ExecuteReader();

//Get RecordsAffected
RecordsAffected = nwReader.RecordsAffected;
```

Listing 19.62 Using the RecordsAffected Property (Visual Basic.NET)

```
dim RecordsAffected as Int32

'Instantiate and open connection object
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

'Get command object
dim nwCmd as new SqlCommand("SELECT EmployeeID, LastName, _
    FirstName FROM Employees", nwConn)

'Execute reader
dim nwReader as object
nwReader = nwCmd.ExecuteReader()

'Get RecordsAffected
RecordsAffected = nwReader.RecordsAffected
```

SqlDataReader.Close()

Syntax

Void Close()

Description

The *Close()* method closes the *SqlDataReader* object.

Example

Listings 19.63 and 19.64 demonstrate how to close a data reader.

Listing 19.63 Closing the SqlDataReader Object (C#)

```
SqlDataReader nwReader = nwCmd.ExecuteReader();
nwReader.Close();
```

Listing 19.64 Closing the SqlDataReader Object (Visual Basic.NET)

```
dim nwReader as object
nwReader = nwCmd.ExecuteReader()
nwReader.Close()
```

SqlDataReader.GetBoolean()

Syntax

```
Boolean GetBoolean( Int32 i )
```

Description

The GetBoolean() method returns the value of a specified column as type Boolean.

Example

Listings 19.65 and 19.66 assume that the value being returned from a query is the correct type.

Listing 19.65 Retrieving Database Values with GetBoolean() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

Boolean value;

while (nwReader.Read()) {

    value = nwReader.GetBoolean(0);
}

nwReader.Close();
```

Listing 19.66 Retrieving Database Values with GetBoolean() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()
```

Listing 19.66 continued

```
dim value as Boolean

while nwReader.Read()
    value = nwReader.GetBoolean(0)
end while

nwReader.Close()
```

SqlDataReader.GetByte()

Syntax

```
Byte GetByte( Int32 i )
```

Description

The `GetByte()` method returns the value of a specified column as type `Byte`.

Example

Listings 19.67 and 19.68 assume that the value being returned from a query is of the correct type.

Listing 19.67 Retrieving Database Values with GetByte() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

Byte value;

while (nwReader.Read()) {

    value = nwReader.GetByte(0);
}

nwReader.Close();
```

Listing 19.68 Retrieving Database Values with GetByte() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()
```

Listing 19.68 continued

```

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as Byte

while nwReader.Read()
    value = nwReader.GetByte(0)
end while

nwReader.Close()

```

SqlDataReader.GetBytes()

Syntax

```

Int32 GetBytes( Int32 i, Int32 dataIndex, Byte[] buffer,
                Int32 bufferIndex, Int32 length )

```

Description

The `GetBoolean()` method returns the value of a specified column as type `Boolean`.

Example

Listings 19.69 and 19.70 assume that the value being returned from a query is the correct type.

Listing 19.69 Retrieving Database Values with GetBytes() (C#)

```

SqlConnection nwConn = new SqlConnection("Data Source=localhost;
                                         User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

Byte[] value = new Byte[10];

while (nwReader.Read()) {
    nwReader.GetBytes(0, 0, value, 0, 10);
}

nwReader.Close();

```

Listing 19.70 Retrieving Database Values with GetBytes() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
    User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)  
  
dim nwReader as object  
nwReader = nwCmd.ExecuteReader()  
  
dim value as Byte(10)  
  
while nwReader.Read()  
    value = nwReader.GetBytes(0)  
end while  
  
nwReader.Close()
```

SqlDataReader.GetChar()

Syntax

Char GetChar(Int32 i)

Description

The *GetChar()* method returns the value of a specified column as type *Char*.

Example

Listings 19.71 and 19.72 assume that the value being returned from a query is of the correct type.

Listing 19.71 Retrieving Database Values with GetChar() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
    User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);  
  
SqlDataReader nwReader = nwCmd.ExecuteReader();  
  
char value;  
  
while (nwReader.Read()) {
```

Listing 19.71 continued

```

    value = nwReader.GetChar(0);
}

nwReader.Close();

```

Listing 19.72 Retrieving Database Values with GetChar() (Visual Basic.NET)

```

dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as char

while nwReader.Read()
    value = nwReader.GetChar(0)
end while

nwReader.Close()

```

SqlDataReader.GetChars()

Syntax

```

Int32 GetChars( Int32 i, Int32 dataIndex, Char[] buffer,
                Int32 bufferIndex, Int32 length )

```

Description

The `GetBoolean()` method returns the value of a specified column as type `Boolean`.

Example

Listings 19.73 and 19.74 assume that the value being returned from a query is of the correct type.

Listing 19.73 Retrieving Database Values with GetChars() (C#)

```

SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

```

Listing 19.73 continued

```
SqlDataReader nwReader = nwCmd.ExecuteReader();

char[] value = new char[10];

while (nwReader.Read()) {

    nwReader.GetChars(0, 0, value, 0, 10);
}

nwReader.Close();
```

Listing 19.74 Retrieving Database Values with GetChars() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as char()

while nwReader.Read()
    value = nwReader.GetChars(0)
end while

nwReader.Close()
```

`SqlDataReader.GetDataTypeName()`**Syntax**

```
String GetDataTypeName( Int32 i )
```

Description

The `GetDataTypeName()` method returns a string that contains the data type of the specified field.

Example

Listings 19.75 and 19.76 demonstrate how to return the type name of a field in the data reader.

Listing 19.75 Retrieving the Type Name (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
```

Listing 19.75 continued

```

nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

string dataType = nwReader.GetDataTypeName(0);

nwReader.Close();

```

Listing 19.76 Retrieving the Type Name (Visual Basic.NET)

```

dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT generic FROM Generic_Table", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim dataType as string
dataType = nwReader.GetDataTypeName(0)

nwReader.Close()

```

SqlDataReader.GetDateTime()**Syntax**

DateTime GetDateTime(Int32 i)

Description

The `GetDateTime()` method returns the value of a specified column as type `DateTime`.

Example

Listings 19.77 and 19.78 assume that the value being returned from a query is of the correct type.

Listing 19.77 Retrieving Database Values with GetDateTime() (C#)

```

SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

```

Listing 19.77 continued

```
DateTime value;

while (nwReader.Read()) {

    value = nwReader.GetDateTime(0);
}

nwReader.Close();
```

Listing 19.78 Retrieving Database Values with GetDateTime() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as DateTime

while nwReader.Read()
    value = nwReader.GetDateTime(0)
end while

nwReader.Close()
```

SqlDataReader.GetDecimal()

Syntax

Decimal GetDecimal(*Int32 i*)

Description

The `GetDecimal()` method returns the value of a specified column as type `Decimal`.

Example

Listings 19.79 and 19.80 assume that the value being returned from a query is of the correct type.

Listing 19.79 Retrieving Database Values with GetDecimal() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();
```

Listing 19.79 continued

```
SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

decimal value;

while (nwReader.Read()) {

    value = nwReader.GetDecimal(0);
}

nwReader.Close();
```

Listing 19.80 Retrieving Database Values with GetDecimal() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as decimal

while nwReader.Read()
    value = nwReader.GetDecimal(0)
end while

nwReader.Close()
```

SqlDataReader.GetDouble()

Syntax

Double GetDouble(Int32 i)

Description

The *GetDouble()* method returns the value of a specified column as type *Double*.

Example

Listings 19.81 and 19.82 assume that the value being returned from a query is of the correct type.

Listing 19.81 Retrieving Database Values with GetDouble() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);  
  
SqlDataReader nwReader = nwCmd.ExecuteReader();  
  
double value;  
  
while (nwReader.Read()) {  
  
    value = nwReader.GetDouble(0);  
}  
  
nwReader.Close();
```

Listing 19.82 Retrieving Database Values with GetDouble() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
    User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)  
  
dim nwReader as object  
nwReader = nwCmd.ExecuteReader()  
  
dim value as double  
  
while nwReader.Read()  
    value = nwReader.GetDouble(0)  
end while  
  
nwReader.Close()
```

SqlDataReader.GetFieldType()

Syntax

Type *GetFieldType(Int32 i)*

Description

The *GetFieldType()* method returns the type of the specified field.

Example

Listings 19.83 and 19.84 demonstrate how to use `GetFieldType()` to return the actual type of a field.

Listing 19.83 Retrieving the Field Type (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

Type dataType = nwReader.GetFieldType(0);

nwReader.Close();
```

Listing 19.84 Retrieving the Field Type (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT generic FROM Generic_Table", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim dataType as Type
dataType = nwReader.GetFieldType(0)

nwReader.Close()
```

SqlDataReader.GetFloat()

Syntax

```
Single GetFloat( Int32 i )
```

Description

The `GetFloat()` method returns the value of a specified column as type `Float`.

Example

Listings 19.85 and 19.86 assume that the value being returned from a query is of the correct type.

Listing 19.85 Retrieving Database Values with GetFloat() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);  
  
SqlDataReader nwReader = nwCmd.ExecuteReader();  
  
float value;  
  
while (nwReader.Read()) {  
  
    value = nwReader.GetFloat(0);  
}  
  
nwReader.Close();
```

Listing 19.86 Retrieving Database Values with GetFloat() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
    User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)  
  
dim nwReader as object  
nwReader = nwCmd.ExecuteReader()  
  
dim value as Float  
  
while nwReader.Read()  
    value = nwReader.GetFloat(0)  
end while  
  
nwReader.Close()
```

SqlDataReader.GetGuid()

Syntax

```
Guid GetGuid( Int32 i )
```

Description

The `GetGuid()` method returns the value of a specified column as type `Guid`.

Example

Listings 19.87 and 19.88 assume that the value being returned from a query is of the correct type.

Listing 19.87 Retrieving Database Values with GetGuid() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

Guid value;

while (nwReader.Read()) {

    value = nwReader.GetGuid(0);
}

nwReader.Close();
```

Listing 19.88 Retrieving Database Values with GetGuid() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as Guid

while nwReader.Read()
    value = nwReader.GetGuid(0)
end while

nwReader.Close()
```

SqlDataReader.GetInt16()

Syntax

Int16 GetInt16(Int32 i)

Description

The `GetInt16()` method returns the value of a specified column as type `Int16`.

Example

Listings 19.89 and 19.90 assume that the value being returned from a query is of the correct type.

Listing 19.89 Retrieving Database Values with GetInt16() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

Int16 value;

while (nwReader.Read()) {

    value = nwReader.GetInt16(0);
}

nwReader.Close();
```

Listing 19.90 Retrieving Database Values with GetInt16() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as Int16

while nwReader.Read()
    value = nwReader.GetInt16(0)
end while

nwReader.Close()
```

SqlDataReader.GetInt32()

Syntax

```
Int32 GetInt32( Int32 i )
```

Description

The `GetInt32()` method returns the value of a specified column as type `Int32`.

Example

Listings 19.91 and 19.92 assume that the value being returned from a query is of the correct type.

Listing 19.91 Retrieving Database Values with GetInt32() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

Int32 value;

while (nwReader.Read()) {

    value = nwReader.GetInt32(0);
}

nwReader.Close();
```

Listing 19.92 Retrieving Database Values with GetInt32() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as Int32

while nwReader.Read()
    value = nwReader.GetInt32(0)
```

Listing 19.92 continued

```
end while  
  
nwReader.Close()
```

SqlDataReader.GetInt64()

Syntax

```
Int64 GetInt64( Int32 i )
```

Description

The `GetInt64()` method returns the value of a specified column as type `Int64`.

Example

Listings 19.93 and 19.94 assume that the value being returned from a query is of the correct type.

Listing 19.93 Retrieving Database Values with GetInt64() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);  
  
SqlDataReader nwReader = nwCmd.ExecuteReader();  
  
Int64 value;  
  
while (nwReader.Read()) {  
  
    value = nwReader.GetInt64(0);  
}  
  
nwReader.Close();
```

Listing 19.94 Retrieving Database Values with GetInt64() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)  
  
dim nwReader as object  
nwReader = nwCmd.ExecuteReader()
```

Listing 19.94 continued

```
dim value as Int64

while nwReader.Read()
    value = nwReader.GetInt64(0)
end while

nwReader.Close()
```

SqlDataReader.GetName()**Syntax**

```
String GetName( Int32 i )
```

Description

The GetName() method returns a string that contains the name of the SQL field.

Example

Listings 19.95 and 19.96 demonstrate how to use GetName() to return the name of the database field.

Listing 19.95 Retrieve the Database Field Name (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT EmployeeID, LastName,
    FirstName FROM Employees", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

string fieldName = nwReader.GetName(0);

nwReader.Close();
```

Listing 19.96 Retrieve the Database Field Name (Visual Basic.NET)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

dim nwCmd as new SqlCommand("SELECT EmployeeID, LastName,
    FirstName FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()
```

Listing 19.96 continued

```
dim fieldName as string  
fieldName = nwReader.GetName(0)  
  
nwReader.Close()
```

SqlDataReader.GetOrdinal()

Syntax

```
Int32 GetOrdinal( String name )
```

Description

The `GetOrdinal()` method returns an integer that contains the ordinal number of the specified field.

Example

Listings 19.97 and 19.98 demonstrate how to return the ordinal number of a specified field.

Listing 19.97 Retrieve the Field Ordinal Number (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
SqlCommand nwCmd = new SqlCommand("SELECT EmployeeID,  
LastName, FirstName FROM Employees", nwConn);  
  
SqlDataReader nwReader = nwCmd.ExecuteReader();  
  
int fieldOrdinal = nwReader.GetOrdinal(EmployeeID);  
  
nwReader.Close();
```

Listing 19.98 Retrieve the Field Ordinal Number (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwCmd as new SqlCommand("SELECT EmployeeID, LastName, _  
FirstName FROM Employees", nwConn)  
  
dim nwReader as object  
nwReader = nwCmd.ExecuteReader()
```

Listing 19.98 continued

```
dim fieldOrdinal as Int32  
fieldOrdinal = nwReader.GetOrdinal(EmployeeID)  
  
nwReader.Close()
```

SqlDataReader.GetSchemaTable()

Syntax

```
DataTable GetSchemaTable()
```

Description

The `GetSchemaTable()` method returns a data table that contains database schema for the `SqlCommand` object.

Example

Listings 19.99 and 19.100 demonstrate how to retrieve a data table that contains the schema for a query.

Listing 19.99 Retrieving the Schema Table (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);  
  
SqlDataReader nwReader = nwCmd.ExecuteReader();  
  
DataTable dTable = nwReader.GetSchemaTable();  
  
nwReader.Close();
```

Listing 19.100 Retrieving the Schema Table (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwCmd as new SqlCommand("SELECT generic FROM Generic_Table", nwConn)  
  
dim nwReader as object  
nwReader = nwCmd.ExecuteReader()  
  
dim dTable as object  
dTable = nwReader.GetSchemaTable()  
  
nwReader.Close()
```

SqlDataReader.GetSqlBinary()

Syntax

```
SqlBinary GetSqlBinary( Int32 i )
```

Description

The `GetSqlBinary()` method returns the value of a specified column as type `SqlBinary`.

Example

Listings 19.101 and 19.102 assume that the value being returned from a query is of the correct type.

Listing 19.101 Retrieving Database Values with GetSqlBinary() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlBinary value;

while (nwReader.Read()) {

    value = nwReader.GetSqlBinary(0);
}

nwReader.Close();
```

Listing 19.102 Retrieving Database Values with GetSqlBinary() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as System.Data.SqlTypes.SqlBinary

while nwReader.Read()
    value = nwReader.GetSqlBinary(0)
```

Listing 19.102 continued

```
end while

nwReader.Close()
```

SqlDataReader.GetSqlBit()**Syntax**

```
SqlBit GetSqlBit( Int32 i )
```

Description

The `GetSqlBit()` method returns the value of a specified column as type `SqlBit`.

Example

Listings 19.103 and 19.104 assume that the value being returned from a query is of the correct type.

Listing 19.103 Retrieving Database Values with GetSqlBit() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlBit value;

while (nwReader.Read()) {

    value = nwReader.GetSqlBit(0);
}

nwReader.Close();
```

Listing 19.104 Retrieving Database Values with GetSqlBit() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()
```

Listing 19.104 continued

```
dim value as System.Data.SqlTypes.SqlBit

while nwReader.Read()
    value = nwReader.GetSqlBit(0)
end while

nwReader.Close()
```

SqlDataReader.GetSqlByte()

Syntax

```
SqlByte GetSqlByte( Int32 i )
```

Description

The `GetSqlByte()` method returns the value of a specified column as type `SqlByte`.

Example

Listings 19.105 and 19.106 assume that the value being returned from a query is of the correct type.

Listing 19.105 Retrieving Database Values with GetSqlByte() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlByte value;

while (nwReader.Read()) {

    value = nwReader.GetSqlByte(0);
}

nwReader.Close();
```

Listing 19.106 Retrieving Database Values with GetSqlByte() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)
```

```

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as System.Data.SqlTypes.SqlByte

while nwReader.Read()
    value = nwReader.GetSqlByte(0)
end while

nwReader.Close()

```

SqlDataReader.GetSqlDateTime()

Syntax

```
SqlDateTime GetSqlDateTime( Int32 i )
```

Description

The `GetSqlDateTime()` method returns the value of a specified column as type `SqlDateTime`.

Example

Listings 19.107 and 19.108 assume that the value being returned from a query is of the correct type.

Listing 19.107 Retrieving Database Values with GetSqlDateTime() (C#)

```

SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlDateTime value;

while (nwReader.Read()) {

    value = nwReader.GetSqlDateTime(0);
}

nwReader.Close();

```

Listing 19.108 Retrieving Database Values with GetSqlDateTime() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
```

Listing 19.108 continued

```
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as System.Data.SqlTypes.SqlDateTime

while nwReader.Read()
    value = nwReader.GetSqlDateTime(0)
end while

nwReader.Close()
```

SqlDataReader.GetSqlDecimal()

Syntax

```
SqlDecimal GetSqlDecimal( Int32 i )
```

Description

The `GetSqlDecimal()` method returns the value of a specified column as type `SqlDecimal`.

Example

Listings 19.109 and 19.110 assume that the value being returned from a query is of the correct type.

Listing 19.109 Retrieving Database Values with GetSqlDecimal() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlDecimal value;

while (nwReader.Read()) {

    value = nwReader.GetSqlDecimal(0);
}

nwReader.Close();
```

Listing 19.110 Retrieving Database Values with GetSqlDecimal() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as System.Data.SqlTypes.SqlDecimal

while nwReader.Read()
    value = nwReader.GetSqlDecimal(0)
end while

nwReader.Close()
```

SqlDataReader.GetSqlDouble()

Syntax

```
SqlDouble GetSqlDouble( Int32 i )
```

Description

The GetSqlDouble() method returns the value of a specified column as type SqlDouble.

Example

Listings 19.111 and 19.112 assume that the value being returned from a query is of the correct type.

Listing 19.111 Retrieving Database Values with GetSqlDouble() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlDouble value;

while (nwReader.Read()) {

    value = nwReader.GetSqlDouble(0);
}

nwReader.Close();
```

Listing 19.112 Retrieving Database Values with GetSqlDouble() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
    User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)  
  
dim nwReader as object  
nwReader = nwCmd.ExecuteReader()  
  
dim value as System.Data.SqlTypes.SqlDouble  
  
while nwReader.Read()  
    value = nwReader.GetSqlDouble(0)  
end while  
  
nwReader.Close()
```

SqlDataReader.GetSqlGuid()

Syntax

```
SqlGuid GetSqlGuid( Int32 i )
```

Description

The `GetSqlGuid()` method returns the value of a specified column as type `SqlGuid`.

Example

Listings 19.113 and 19.114 assume that the value being returned from a query is of the correct type.

Listing 19.113 Retrieving Database Values with GetSqlGuid() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
    User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);  
  
SqlDataReader nwReader = nwCmd.ExecuteReader();  
  
System.Data.SqlTypes.SqlGuid value;  
  
while (nwReader.Read()) {  
  
    value = nwReader.GetSqlGuid(0);  
}  
  
nwReader.Close();
```

Listing 19.114 Retrieving Database Values with GetSqlGuid() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as System.Data.SqlTypes.SqlGuid

while nwReader.Read()
    value = nwReader.GetSqlGuid(0)
end while

nwReader.Close()
```

SqlDataReader.GetInt16()

Syntax

```
SqlInt16 GetSqlInt16( Int32 i )
```

Description

The `GetSqlInt16()` method returns the value of a specified column as type `SqlInt16`.

Example

Listings 19.115 and 19.116 assume that the value being returned from a query is of the correct type.

Listing 19.115 Retrieving Database Values with GetSqlInt16() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlInt16 value;

while (nwReader.Read()) {

    value = nwReader.GetSqlInt16(0);
}

nwReader.Close();
```

Listing 19.116 Retrieving Database Values with GetSqlInt16() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
    User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)  
  
dim nwReader as object  
nwReader = nwCmd.ExecuteReader()  
  
dim value as System.Data.SqlTypes.SqlInt16  
  
while nwReader.Read()  
    value = nwReader.GetSqlInt16(0)  
end while  
  
nwReader.Close()
```

SqlDataReader.GetInt32()

Syntax

```
SqlInt32 GetSqlInt32( Int32 i )
```

Description

The GetSqlInt32() method returns the value of a specified column as type SqlInt32.

Example

Listings 19.117 and 19.118 assume that the value being returned from a query is of the correct type.

Listing 19.117 Retrieving Database Values with GetSqlInt32() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
    User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);  
  
SqlDataReader nwReader = nwCmd.ExecuteReader();  
  
System.Data.SqlTypes.SqlInt32 value;  
  
while (nwReader.Read()) {  
  
    value = nwReader.GetSqlInt32(0);  
}  
  
nwReader.Close();
```

Listing 19.118 Retrieving Database Values with GetSqlInt32() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as System.Data.SqlTypes.SqlInt32

while nwReader.Read()
    value = nwReader.GetSqlInt32(0)
end while

nwReader.Close()
```

SqlDataReader.GetInt64()

Syntax

```
SqlInt64 GetSqlInt64( Int32 i )
```

Description

The `GetSqlInt64()` method returns the value of a specified column as type `SqlInt64`.

Example

Listings 19.119 and 19.120 assume that the value being returned from a query is of the correct type.

Listing 19.119 Retrieving Database Values with GetSqlInt64() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlInt64 value;

while (nwReader.Read()) {

    value = nwReader.GetSqlInt64(0);
}

nwReader.Close();
```

Listing 19.120 Retrieving Database Values with GetSqlInt64() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _  
    User Id=sa; Password=;Initial Catalog=northwind")  
nwConn.Open()  
  
dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)  
  
dim nwReader as object  
nwReader = nwCmd.ExecuteReader()  
  
dim value as System.Data.SqlTypes.SqlInt64  
  
while nwReader.Read()  
    value = nwReader.GetSqlInt64(0)  
end while  
  
nwReader.Close()
```

SqlDataReader.GetSqlMoney()

Syntax

```
SqlMoney GetSqlMoney( Int32 i )
```

Description

The GetSqlMoney() method returns the value of a specified column as type SqlMoney.

Example

Listings 19.121 and 19.122 assume that the value being returned from a query is of the correct type.

Listing 19.121 Retrieving Database Values with GetSqlMoney() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;  
    User Id=sa; Password=;Initial Catalog=northwind");  
nwConn.Open();  
  
SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);  
  
SqlDataReader nwReader = nwCmd.ExecuteReader();  
  
System.Data.SqlTypes.SqlMoney value;  
  
while (nwReader.Read()) {  
  
    value = nwReader.GetSqlMoney(0);  
}  
  
nwReader.Close();
```

Listing 19.122 Retrieving Database Values with GetSqlMoney() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as System.Data.SqlTypes.SqlMoney

while nwReader.Read()
    value = nwReader.GetSqlMoney(0)
end while

nwReader.Close()
```

SqlDataReader.GetSqlSingle()

Syntax

```
SqlSingle GetSqlSingle( Int32 i )
```

Description

The GetSqlSingle() method returns the value of a specified column as type SqlSingle.

Example

Listings 19.123 and 19.124 assume that the value being returned from a query is of the correct type.

Listing 19.123 Retrieving Database Values with GetSqlSingle() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlSingle value;

while (nwReader.Read()) {
    value = nwReader.GetSqlSingle(0);
```

Listing 19.123 continued

```
}

nwReader.Close();
```

Listing 19.124 Retrieving Database Values with GetSqlSingle() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as System.Data.SqlTypes.SqlSingle

while nwReader.Read()
    value = nwReader.GetSqlSingle(0)
end while

nwReader.Close()
```

SqlDataReader.GetSqlString()

Syntax

```
SqlString GetSqlString( Int32 i )
```

Description

The `GetSqlString()` method returns the value of a specified column as type `SqlString`.

Example

Listings 19.125 and 19.126 assume that the value being returned from a query is of the correct type.

Listing 19.125 Retrieving Database Values with GetSqlString() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlString value;
```

Listing 19.125 continued

```
while (nwReader.Read()) {
    value = nwReader.GetSqlString(0);
}
nwReader.Close();
```

Listing 19.126 Retrieving Database Values with GetSqlString() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as System.Data.SqlTypes.SqlString

while nwReader.Read()
    value = nwReader.GetSqlString(0)
end while

nwReader.Close()
```

SqlDataReader.GetValue()**Syntax**

Object GetValue(Int32 i)

Description

The `GetValue()` method returns a value of type `Object`, by using its native SQL data type.

Example

Listings 19.127 and 19.128 assume that the value being returned from a query is of the correct type.

Listing 19.127 Retrieving Database Values with GetValue() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();
```

Listing 19.127 continued

```
SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

System.Data.SqlTypes.SqlString value;

while (nwReader.Read()) {
    value = nwReader.GetSqlValue(0);
}

nwReader.Close();
```

Listing 19.128 Retrieving Database Values with GetSqlValue() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT generic FROM Generic_Table", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as System.Data.SqlTypes.SqlString

while nwReader.Read()
    value = nwReader.GetSqlValue(0)
end while

nwReader.Close()
```

SqlDataReader.GetString()

Syntax

```
String GetString( Int32 i )
```

Description

The `GetString()` method returns the value of a specified column as type `String`.

Example

Listings 19.129 and 19.130 assume that the value being returned from a query is of the correct type.

Listing 19.129 Retrieving Database Values with GetString() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

string value;

while (nwReader.Read()) {
    value = nwReader.GetString(0);
}

nwReader.Close();
```

Listing 19.130 Retrieving Database Values with GetString() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT EmployeeID FROM Employees", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as string

while nwReader.Read()
    value = nwReader.GetString(0)
end while

nwReader.Close()
```

SqlDataReader.GetValue()

Syntax

Object GetValue(Int32 i)

Description

The *GetValue()* method returns a value of type *Object*, by using the Microsoft .NET framework types.

Example

Listings 19.131 and 19.132 assume that the value being returned from a query is of the correct type.

Listing 19.131 Retrieving Database Values with GetValue() (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

string value;

while (nwReader.Read()) {
    value = nwReader.GetValue(0);
}

nwReader.Close();
```

Listing 19.132 Retrieving Database Values with GetValue() (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT generic FROM Generic_Table", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

string value

while nwReader.Read()
    value = nwReader.GetValue(0)
end while

nwReader.Close()
```

SqlDataReader.IsDBNull()

Syntax

```
Boolean IsDBNull( Int32 i )
```

Description

The `IsDBNull()` method returns True if the specified column is null and False otherwise.

Example

Listings 19.133 and 19.134 demonstrate how to use `IsDBNull()` to detect a null field.

Listing 19.133 Detecting Null Fields (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("SELECT generic FROM Generic_Table", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

string value;

while (nwReader.Read()) {
    if( nwReader.IsDBNull(0) != true )
        value = nwReader.GetValue(0);
}

nwReader.Close();
```

Listing 19.134 Detecting Null Fields (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("SELECT generic FROM Generic_Table", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as string

while nwReader.Read()
    if nwReader.IsDBNull(0) <> true
        value = nwReader.GetValue(0)
    end if
end while

nwReader.Close()
```

SqlDataReader.NextResult()

Syntax

```
Boolean NextResult()
```

Description

The `NextResult()` method advances the data reader to the next record. It is used when reading the result of SQL batch statements.

Example

Listings 19.135 and 19.136 demonstrate how to check for the existence of more rows in a data reader.

Listing 19.135 Using the NextResult() Method (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("exec SqlBatch", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

string value;

while (nwReader.NextResult()) {
    value = nwReader.GetValue(0);
}

nwReader.Close();
```

Listing 19.136 Using the NextResult() Method (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("exec SqlBatch", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as string

while nwReader.NextResult()
    value = nwReader.GetValue(0)
end while

nwReader.Close()
```

SqlDataReader.Read()

Syntax

```
Boolean Read()
```

Description

The `Read()` method advances the data reader to the next record. It returns True if there are more records and False otherwise.

Example

Listings 19.137 and 19.138 demonstrate how to use the `Read()` method to step through a result set.

Listing 19.137 Reading the Next Record (C#)

```
SqlConnection nwConn = new SqlConnection("Data Source=localhost;
    User Id=sa; Password=;Initial Catalog=northwind");
nwConn.Open();

SqlCommand nwCmd = new SqlCommand("exec SqlBatch", nwConn);

SqlDataReader nwReader = nwCmd.ExecuteReader();

string value;

while (nwReader.NextResult()) {
    value = nwReader.GetValue(0);
}

nwReader.Close();
```

Listing 19.138 Reading the Next Record (Visual Basic.NET)

```
dim nwConn as new SqlConnection("Data Source=localhost; _
    User Id=sa; Password=;Initial Catalog=northwind")
nwConn.Open()

dim nwCmd as new SqlCommand("exec SqlBatch", nwConn)

dim nwReader as object
nwReader = nwCmd.ExecuteReader()

dim value as string

while nwReader.NextResult()
    value = nwReader.GetValue(0)
end while

nwReader.Close()
```

The SqlDbType Class

The SqlDbType class contains a set of static constants that mirror the data types found in SQL server versions 7.0 and above (see Table 19.4).

Table 19.4 Properties of the SqlDbType Class

Item	Description
BigInt	Contains a BigInt data type
Binary	Contains a Binary data type, with a length of 0 to 8000 bytes
Bit	Contains a Bit data type
Char	Contains a Char data type
DateTime	Contains an 8-byte DateTime data type
Decimal	Contains a Decimal data type
Float	Contains an 8-byte Float data type
Image	Contains an Image data type
Int	Contains a 4-byte Int data type
Money	Contains an 8-byte Money data type
NChar	Contains Unicode character data
NText	Contains Unicode text data
NVarChar	Contains Unicode variable-length character data
Real	Contains a 4-byte Real data type
SmallDateTime	Contains a 4-byte SmallDateTime data type
SmallInt	Contains a 2-byte SmallInt data type
SmallMoney	Contains a 4-byte SmallMoney data type
Text	Contains a Text data type
Timestamp	Contains a Timestamp data type
TinyInt	Contains a 1-byte TinyInt data type
UniqueIdentifier	Contains a UniqueIdentifier data type
VarBinary	Contains variable-length binary data, with a length of 0 to 8,000 bytes
VARCHAR	Contains variable-length character data, with a length of 0 to 8,000 bytes
VARIANT	Contains a Variant data type

The SqlParameter Class

The SqlParameter class is used to pass parameters to a stored procedure in a SQL database. Table 19.5 shows all the properties in the SqlParameter class, along with a brief description of each property.

Table 19.5 Properties of the SqlParameter Class

Item	Description
DbType	Contains the data type of the SQL parameter
Direction	Defines whether the parameter is input or output
IsNullable	Defines whether the parameter can be null

Table 19.5 continued

Item	Description
ParameterName	Contains the name of the parameter
Precision	Contains the precision of the parameter
Scale	Contains the scale of the parameter
Size	Contains the size of the parameter
Value	Contains the value of the parameter

SqlParameter.DbType

Syntax

```
SqlDbType DbType
```

Description

The `DbType` property is set to the SQL data type that is expected by the stored procedure. Any of the values in the `System.Data.SqlClient.SqlDbType` namespace are acceptable.

Example

Listings 19.139 and 19.140 demonstrate how to set the `DbType` property for a parameter.

Listing 19.139 Setting the DbType Property for a Parameter (C#)

```
SqlParameter param = new SqlParameter();
param.DbType = SqlDbType.NVarChar;
```

Listing 19.140 Setting the DbType Property for a Parameter (Visual Basic.NET)

```
dim param as new SqlParameter()
param.DbType = SqlDbType.NVarChar
```

SqlParameter.Direction

Syntax

```
ParameterDirection Direction
```

Description

The `Direction` property sets the direction of the parameter and should match the stored procedure. Acceptable values for this property are found in the `System.Data.SqlClient.ParameterDirection` namespace.

Example

Listings 19.141 and 19.142 demonstrate how to set the `Direction` property for a parameter.

Listing 19.141 Setting the Direction Property for a Parameter (C#)

```
SqlParameter param = new SqlParameter();  
  
param.Direction = ParameterDirection.Input;
```

Listing 19.142 Setting the Direction Property for a Parameter (Visual Basic.NET)

```
dim param as new SqlParameter()  
  
param.Direction = ParameterDirection.Input
```

SqlParameter.IsDBNull

Syntax

```
Boolean IsNullable
```

Description

The `IsNullable` property specifies whether the parameter is allowed to be null.

Example

Listings 19.143 and 19.144 demonstrate how to set the `IsNullable` property of a parameter.

Listing 19.143 Setting the IsNullable Property for a Parameter (C#)

```
SqlParameter param = new SqlParameter();  
  
param.IsNullable = true;
```

Listing 19.144 Setting the IsNullable Property for a Parameter (Visual Basic.NET)

```
dim param as new SqlParameter()  
  
param.IsNullable = true
```

SqlParameter.ParameterName

Syntax

```
String ParameterName
```

Description

The ParameterName property specifies the name of the parameter that the stored procedure is expecting.

Example

Listings 19.145 and 19.146 demonstrate how to set the ParameterName property of a parameter.

Listing 19.145 Specifying the ParameterName Property of a Parameter (C#)

```
param.ParameterName = "@CustomerID";
```

Listing 19.146 Specifying the ParameterName Property of a Parameter (Visual Basic.NET)

```
param.ParameterName = "@CustomerID"
```

SqlParameter.Precision

Syntax

Byte Precision

Description

The Precision property is used to specify the number of digits used to represent the value of the parameter.

Example

Listings 19.147 and 19.148 demonstrate how to specify the Precision property of a parameter.

Listing 19.147 Specifying the Precision Property of a Parameter (C#)

```
SqlParameter param = new SqlParameter();
param.Precision = 2;
```

Listing 19.148 Specifying the Precision Property of a Parameter (Visual Basic.NET)

```
dim param as new SqlParameter()
param.Precision = 2;
```

SqlParameter.Scale

Syntax

Byte Scale

Description

The Scale property is used to specify the number of decimal places in the value of the parameter.

Example

Listings 19.149 and 19.150 demonstrate how to specify the Scale property of a parameter.

Listing 19.149 Specifying the Scale property of a parameter (C#)

```
SqlParameter param = new SqlParameter();
param.Scale = 2;
```

Listing 19.150 Specifying the Scale property of a parameter (Visual Basic.NET)

```
dim param as new SqlParameter()
param.Scale = 2
```

SqlParameter.Size

Syntax

Int32 Size

Description

The Size property specifies the size of the parameter that the stored procedure is expecting.

Example

Listings 19.151 and 19.152 demonstrate how to set the Size property of a parameter.

Listing 19.151 Specifying the Size Property of a Parameter (C#)

```
SqlParameter param = new SqlParameter();
param.Size = 5;
```

Listing 19.152 Specifying the Size Property of a Parameter (Visual Basic.NET)

```
dim param as new SqlParameter()
param.Size = 5
```

SqlParameter.Value

Syntax

Object Value

Description

The Value property is used to specify the value of the parameter that will be passed to the stored procedure.

Example

Listings 19.153 and 19.154 demonstrate how to set the Value property of a parameter.

Listing 19.153 Setting the Value Property of a Parameter (C#)

```
SqlParameter param = new SqlParameter()
param.Value = "Value"
```

Listing 19.154 Setting the Value Property of a Parameter (Visual Basic.NET)

```
dim param as new SqlParameter()
param.Value = "Value"
```

The SqlParameterCollection Class

The SqlParameterCollection class contains all the properties and methods that are necessary for working with SQL parameters (see Table 19.6).

Table 19.6 Properties and Methods of the SqlParameterCollection Class

Item	Description
Properties	
Count	Contains the number of items in the parameter collection
Item	Provides an index for the collection
Methods	
Add()	Adds a new parameter to the parameter collection
Clear()	Removes all items from the collection
Contains()	Enables a developer to check for the existence of a specific parameter in the collection
IndexOf()	Returns the location of a specific parameter in the collection
Insert()	Adds a parameter to the collection at a specific place
Remove()	Removes a parameter from the collection
RemoveAt()	Removes a specific parameter from the collection by ordinal number or name

SqlParameterCollection.Count

Syntax

Int32 Count

Description

Count is a read-only property that contains the number of items in the parameter collection.

Example

Listings 19.155 and 19.156 demonstrate how to determine the number of items in a parameter collection.

Listing 19.155 Retrieving the Number of Items in a Collection (C#)

```
//Create and Open Connection
SqlConnection nwConn = new SqlConnection("Data Source=localhost;Initial
    Catalog=northwind;UID=sa;PWD=");
nwConn.Open();

//Create Command
SqlCommand nwCmd = new SqlCommand("AddNewCustomer", nwConn);
nwCmd.CommandType = CommandType.StoredProcedure;

//Declare and configure parameter
SqlParameter param = nwCmd.CreateParameter();
param.ParameterName = "@CustomerID";
param.DbType = SqlDbType.NVarChar;
param.Size = 5;
param.Value = "12345";
param.Direction = ParameterDirection.Input;

int count = nwCmd.Parameters.Count;
```

Listing 19.156 Retrieving the Number of Items in a Collection (Visual Basic.NET)

```
'Create and Open Connection
dim nwConn as new SqlConnection("Data Source=localhost;Initial _
    Catalog=northwind;UID=sa;PWD;")
nwConn.Open()

'Create Command
dim nwCmd as new SqlCommand("AddNewCustomer", nwConn)
nwCmd.CommandType = CommandType.StoredProcedure

'Declare and configure parameter
dim param as object
dim param = nwCmd.CreateParameter()
param.ParameterName = "@CustomerID"
param.DbType = SqlDbType.NVarChar
param.Size = 5
param.Value = "12345"
```

Listing 19.156 continued

```
param.Direction = ParameterDirection.Input  
  
dim count as Int32  
count = nwCmd.Parameters.Count
```

SqlParameterCollection.Item

Syntax

```
SqlParameter Item( String parameterName )  
SqlParameter Item( Int32 index )
```

Description

The Item property is used to access a specific SQL parameter in the collection, by name or by index.

Example

Listings 19.157 and 19.158 demonstrate how to access specific SQL parameters in the parameter collection.

Listing 19.157 Accessing a Specific SQL Parameter (C#)

```
//Create and Open Connection  
SqlConnection nwConn = new SqlConnection("Data Source=localhost;Initial  
Catalog=northwind;UID=sa;PWD=");  
nwConn.Open();  
  
//Create Command  
SqlCommand nwCmd = new SqlCommand("AddNewCustomer", nwConn);  
nwCmd.CommandType = CommandType.StoredProcedure;  
  
//Declare and configure parameter  
SqlParameter param = nwCmd.CreateParameter();  
param.ParameterName = "@CustomerID";  
param.DbType = SqlDbType.NVarChar;  
param.Size = 5;  
param.Value = "12345";  
param.Direction = ParameterDirection.Input;  
  
nwCmd.Parameters.Add(param);  
param2 = nwCmd.Parameters["@CustomerID"];  
param3 = nwCmd.Parameters[0];
```

Listing 19.158 Accessing a Specific SQL Parameter (Visual Basic.NET)

```
'Create and Open Connection  
dim nwConn as new SqlConnection("Data Source=localhost;Initial _  
Catalog=northwind;UID=sa;PWD=")
```

Listing 19.158 continued

```
nwConn.Open()

'Create Command
dim nwCmd as new SqlCommand("AddNewCustomer", nwConn)
nwCmd.CommandType = CommandType.StoredProcedure

'Declare and configure parameter
dim param as object
dim param = nwCmd.CreateParameter()
param.ParameterName = "@CustomerID"
param.DbType = SqlDbType.NVarChar
param.Size = 5
param.Value = "12345"
param.Direction = ParameterDirection.Input

nwCmd.Parameters.Add(param)
param2 = nwCmd.Parameters("@CustomerID")
param3 = nwCmd.Parameters(0)
```

SqlParameterCollection.Add()

Syntax

```
SqlParameter Add( String name, SqlDbType dbType )
SqlParameter Add( ISqlParameter value )
Int32 Add( Object value )
SqlParameter Add( String name, Object value )
SqlParameter Add( String name, SqlDbType dbType, Int32 size,
                 String sourceColumn )
SqlParameter Add( String name, SqlDbType dbType, Int32 size )
```

Description

The Add() method adds SQL parameters to the parameter collection.

Example

Listings 19.159 and 19.160 demonstrate how to add various parameters to the parameter collection.

Listing 19.159 Adding Parameters to the Parameter Collection (C#)

```
//Create and Open Connection
SqlConnection nwConn = new SqlConnection("Data Source=localhost;Initial _ 
                                         Catalog=northwind;UID=sa;PWD=;");
nwConn.Open();

//Create Command
SqlCommand nwCmd = new SqlCommand("AddNewCustomer", nwConn);
nwCmd.CommandType = CommandType.StoredProcedure;
```

Listing 19.159 continued

```
//Declare and configure parameter
SqlParameter param = nwCmd.CreateParameter();
param.ParameterName = "@CustomerID";
param.DbType = SqlDbType.NVarChar;
param.Size = 5;
param.Value = "12345";
param.Direction = ParameterDirection.Input;

nwCmd.Parameters.Add(param);
```

Listing 19.160 Adding Parameters to the Parameter Collection (Visual Basic.NET)

```
'Create and Open Connection
dim nwConn as new SqlConnection("Data Source=localhost;Initial _
    Catalog=northwind;UID=sa;PWD=;")
nwConn.Open()

'Create Command
dim nwCmd as new SqlCommand("AddNewCustomer", nwConn)
nwCmd.CommandType = CommandType.StoredProcedure

'Declare and configure parameter
dim param as object
dim param = nwCmd.CreateParameter()
param.ParameterName = "@CustomerID"
param.DbType = SqlDbType.NVarChar
param.Size = 5
param.Value = "12345"
param.Direction = ParameterDirection.Input

nwCmd.Parameters.Add(param)
```

SqlParameterCollection.Clear()

Syntax

```
Void Clear()
```

Description

The `Clear()` method removes all items from the parameter collection.

Example

Listings 19.161 and 19.162 demonstrate how to use the `Clear()` method.

Listing 19.161 Using the Clear() Method (C#)

```
//Create and Open Connection
SqlConnection nwConn = new SqlConnection("Data Source=localhost;Initial
```

Listing 19.161 continued

```
Catalog=northwind;UID=sa;PWD=");  
nwConn.Open();  
  
//Create Command  
SqlCommand nwCmd = new SqlCommand("AddNewCustomer", nwConn);  
nwCmd.CommandType = CommandType.StoredProcedure;  
  
nwCmd.Parameters.Clear();
```

Listing 19.162 Using the clear() Method (Visual Basic.NET)

```
'Create and Open Connection  
dim nwConn as new SqlConnection("Data Source=localhost;Initial _  
    Catalog=northwind;UID=sa;PWD=")  
nwConn.Open()  
  
'Create Command  
dim nwCmd as new SqlCommand("AddNewCustomer", nwConn)  
nwCmd.CommandType = CommandType.StoredProcedure  
  
nwCmd.Parameters.Clear()
```

SqlParameterCollection.Contains()

Syntax

```
Boolean Contains( String value )  
Boolean Contains( Object value )
```

Description

The Contains() method returns True if the specified items are found in the parameter collection.

Example

Listings 19.163 and 19.164 demonstrate how to use the Contains() method.

Listing 19.163 Using the Contains() Method (C#)

```
//Create and Open Connection  
SqlConnection nwConn = new SqlConnection("Data Source=localhost;Initial _  
    Catalog=northwind;UID=sa;PWD=");  
nwConn.Open();  
  
//Create Command  
SqlCommand nwCmd = new SqlCommand("AddNewCustomer", nwConn);  
nwCmd.CommandType = CommandType.StoredProcedure;  
  
//Declare and configure parameter
```

Listing 19.163 continued

```

SqlParameter param = nwCmd.CreateParameter();
param.ParameterName = "@CustomerID";
param.DbType = SqlDbType.NVarChar;
param.Size = 5;
param.Value = "12345";
param.Direction = ParameterDirection.Input;

nwCmd.Parameters.Add(param);

Bool IsPresent = nwCmd.Parameters.Contains( param );
Bool IsPresent2 = nwCmd.Parameters.Contains("@CustomerID");

```

Listing 19.164 Using the Contains() Method (Visual Basic.NET)

```

'Create and Open Connection
dim nwConn as new SqlConnection("Data Source=localhost;Initial _
    Catalog=northwind;UID=sa;PWD=")
nwConn.Open()

'Create Command
dim nwCmd as new SqlCommand("AddNewCustomer", nwConn)
nwCmd.CommandType = CommandType.StoredProcedure

'Declare and configure parameter
dim param as object
dim param = nwCmd.CreateParameter()
param.ParameterName = "@CustomerID"
param.DbType = SqlDbType.NVarChar
param.Size = 5
param.Value = "12345"
param.Direction = ParameterDirection.Input

dim IsPresent as Boolean
IsPresent = nwCmd.Parameters.Contains( param )
IsPresent = nwCmd.Parameters.Contains("@CustomerID")

```

SqlParameterCollection.IndexOf()**Syntax**

Int32 IndexOf(Object parameter)
Int32 IndexOf(String parameterName)

Description

The `IndexOf()` method returns the index of the specified parameter in the parameter collection.

Example

Listings 19.165 and 19.166 demonstrate how to use the `IndexOf()` method to find the index of a specific parameter in the parameter collection.

Listing 19.165 Getting the Index of a Parameter (C#)

```
//Create and Open Connection
SqlConnection nwConn = new SqlConnection("Data Source=localhost;Initial
    Catalog=northwind;UID=sa;PWD;");
nwConn.Open();

//Create Command
SqlCommand nwCmd = new SqlCommand("AddNewCustomer", nwConn);
nwCmd.CommandType = CommandType.StoredProcedure;

//Declare and configure parameter
SqlParameter param = nwCmd.CreateParameter();
param.ParameterName = "@CustomerID";
param.DbType = SqlDbType.NVarChar;
param.Size = 5;
param.Value = "12345";
param.Direction = ParameterDirection.Input;

nwCmd.Parameters.Add(param);

int paramIndex = nwCmd.Parameters.IndexOf( param );
int paramIndex2 = nwCmd.Parameters.IndexOf("@CustomerID");
```

Listing 19.166 Getting the Index of a Parameter (Visual Basic.NET)

```
'Create and Open Connection
dim nwConn as new SqlConnection("Data Source=localhost;Initial _
    Catalog=northwind;UID=sa;PWD;")
nwConn.Open()

'Create Command
dim nwCmd as new SqlCommand("AddNewCustomer", nwConn)
nwCmd.CommandType = CommandType.StoredProcedure

'Declare and configure parameter
dim param as object
dim param = nwCmd.CreateParameter()
param.ParameterName = "@CustomerID"
param.DbType = SqlDbType.NVarChar
param.Size = 5
param.Value = "12345"
param.Direction = ParameterDirection.Input
nwCmd.Parameters.Add(param)
```

Listing 19.166 continued

```
dim paramIndex as Int32  
paramIndex = nwCmd.Parameters.IndexOf( param )  
paramIndex = nwCmd.Parameters.IndexOf("@CustomerID")
```

SqlParameterCollection.Insert()

Syntax

```
Void Insert( Int32 index, Object value )
```

Description

The `Insert()` method adds a parameter to the parameter collection at a specific place that is given by index.

Example

Listings 19.167 and 19.168 demonstrate how to use the `Insert()` method to add a parameter to the collection.

Listing 19.167 Inserting a Parameter into the Parameter Collection (C#)

```
//Create and Open Connection  
SqlConnection nwConn = new SqlConnection("Data Source=localhost;Initial  
Catalog=northwind;UID=sa;PWD=");  
nwConn.Open();  
  
//Create Command  
SqlCommand nwCmd = new SqlCommand("AddNewCustomer", nwConn);  
nwCmd.CommandType = CommandType.StoredProcedure;  
  
//Declare and configure parameter  
SqlParameter param = nwCmd.CreateParameter();  
param.ParameterName = "@CustomerID";  
param.DbType = SqlDbType.NVarChar;  
param.Size = 5;  
param.Value = "12345";  
param.Direction = ParameterDirection.Input;  
  
//Insert Parameter  
nwCmd.Parameters.Insert(0, param);
```

***Listing 19.168 Inserting a Parameter into the Parameter Collection
(Visual Basic.NET)***

```
'Create and Open Connection  
dim nwConn as new SqlConnection("Data Source=localhost;Initial _  
Catalog=northwind;UID=sa;PWD=")  
nwConn.Open()
```

Listing 19.168 continued

```
'Create Command
dim nwCmd as new SqlCommand("AddNewCustomer", nwConn)
nwCmd.CommandType = CommandType.StoredProcedure

'Declare and configure parameter
dim param as object
dim param = nwCmd.CreateParameter()
param.ParameterName = "@CustomerID"
param.DbType = SqlDbType.NVarChar
param.Size = 5
param.Value = "12345"
param.Direction = ParameterDirection.Input
nwCmd.Parameters.Add(param)

//Insert Parameter
nwCmd.Parameters.Insert(0, param)
```

SqlParameterCollection.Remove()

Syntax

```
Void Remove( Object value )
```

Description

The `Remove()` method removes a parameter from the collection. The parameter object is passed in as the `Value` argument.

Example

Listings 19.169 and 19.170 demonstrate how to remove a parameter from the parameter collection.

Listing 19.169 Removing a Parameter from the Parameter Collection (C#)

```
//Create and Open Connection
SqlConnection nwConn = new SqlConnection("Data Source=localhost;Initial
    Catalog=northwind;UID=sa;PWD=");
nwConn.Open();

//Create Command
SqlCommand nwCmd = new SqlCommand("AddNewCustomer", nwConn);
nwCmd.CommandType = CommandType.StoredProcedure;

//Declare and configure parameter
SqlParameter param = nwCmd.CreateParameter();
param.ParameterName = "@CustomerID";
param.DbType = SqlDbType.NVarChar;
param.Size = 5;
param.Value = "12345";
```

Listing 19.169 continued

```
param.Direction = ParameterDirection.Input;  
  
//Remove Parameters  
nwCmd.Parameters.Remove(param);
```

***Listing 19.170 Removing a Parameter from the Parameter Collection
(Visual Basic.NET)***

```
'Create and Open Connection  
dim nwConn as new SqlConnection("Data Source=localhost;Initial _  
    Catalog=northwind;UID=sa;PWD=")  
nwConn.Open()  
  
'Create Command  
dim nwCmd as new SqlCommand("AddNewCustomer", nwConn)  
nwCmd.CommandType = CommandType.StoredProcedure  
  
'Declare and configure parameter  
dim param as object  
dim param = nwCmd.CreateParameter()  
param.ParameterName = "@CustomerID"  
param.DbType = SqlDbType.NVarChar  
param.Size = 5  
param.Value = "12345"  
param.Direction = ParameterDirection.Input  
nwCmd.Parameters.Add(param)  
  
//Remove Parameters  
nwCmd.Parameters.Remove(param)
```

SqlParameterCollection.RemoveAt()

Syntax

```
Void RemoveAt( String value )  
Void RemoveAt( Int32 index )
```

Description

The RemoveAt() method removes a parameter from the parameter collection, either by name or by index.

Example

Listings 19.171 and 19.172 demonstrate how to remove items from the parameter name, both by name and by index.

Listing 19.171 Removing Items from the Parameter Collection (C#)

```
//Create and Open Connection  
SqlConnection nwConn = new SqlConnection("Data Source=localhost;Initial
```

Listing 19.171 continued

```
Catalog=northwind;UID=sa;PWD=");  
nwConn.Open();  
  
//Create Command  
SqlCommand nwCmd = new SqlCommand("AddNewCustomer", nwConn);  
nwCmd.CommandType = CommandType.StoredProcedure;  
  
//Add a parameter  
nwParam = nwCmd.Parameters.Add(new SqlParameter("@CompanyName",  
    SqlDbType.NVarChar, 40));  
nwParam.Direction = ParameterDirection.Input;  
nwParam.Value = "test";  
  
//Add a parameter  
nwParam = nwCmd.Parameters.Add(new SqlParameter("@Country",  
    SqlDbType.NVarChar, 15));  
nwParam.Direction = ParameterDirection.Input;  
nwParam.Value = "test";  
  
//Remove Parameters  
nwCmd.Parameters.RemoveAt(0);  
nwCmd.Parameters.Remove("@Country");
```

Listing 19.172 Removing Items from the Parameter Collection (Visual Basic.NET)

```
'Create and Open Connection  
dim nwConn as new SqlConnection("Data Source=localhost;Initial _  
    Catalog=northwind;UID=sa;PWD=")  
nwConn.Open()  
  
'Create Command  
dim nwCmd as new SqlCommand("AddNewCustomer", nwConn)  
nwCmd.CommandType = CommandType.StoredProcedure  
  
'Add a parameter  
dim nwParam as object  
nwParam = nwCmd.Parameters.Add(new SqlParameter("@CompanyName", _  
    SqlDbType.NVarChar, 40))  
nwParam.Direction = ParameterDirection.Input  
nwParam.Value = "test"  
  
'Add a parameter  
nwParam = nwCmd.Parameters.Add(new SqlParameter("@Country", _  
    SqlDbType.NVarChar, 15))  
nwParam.Direction = ParameterDirection.Input  
nwParam.Value = "test"  
  
'Remove Parameters
```

Listing 19.172 continued

```
nwCmd.Parameters.RemoveAt(0)
nwCmd.Parameters.RemoveAt("@Country")
```

The SqlTransaction Class

The `SqlTransaction` class provides a way to manually work with transactions (see Table 19.7).

Table 19.7 Properties and Methods of the `SqlTransaction` Class

Item	Description
Property	
<code>IsolationLevel</code>	Contains the isolation level of the transaction in the current connection; read-only
Methods	
<code>Commit()</code>	Commits the transaction
<code>RollBack()</code>	Rolls back the transaction
<code>Save()</code>	Saves the transaction

`SqlTransaction.IsolationLevel`

Syntax

```
IsolationLevel IsolationLevel
```

Description

The `IsolationLevel` property specifies the transaction isolation level for the connection. Valid values for this property are specified in the `IsolationLevel` class in the `System.Data` namespace. If the value of the `IsolationLevel` property is not specified, the default value is `ReadCommitted`.

Example

Listings 19.173 and 19.174 demonstrate how to use the `BegginTransaction()` method to start a database transaction.

Listing 19.173 Setting the Isolation Level for a Transaction (C#)

```
SqlTransaction nwTrans = nwConn.BeginTransaction(IsolationLevel.Serializable,
    "RemoveEmployees");

msg.Text = nwTrans.IsolationLevel.ToString();
```

Listing 19.174 Setting the Isolation Level for a Transaction (Visual Basic.NET)

```
dim nwTrans as object
nwTrans = nwConn.BeginTransaction(IsolationLevel.Serializable, _
```

Listing 19.174 continued

```
"RemoveEmployees")  
  
msg.Text = nwTrans.IsolationLevel.ToString()
```

SqlTransaction.Commit()

Syntax

```
Void Commit()
```

Description

The `Commit()` method finalizes a transaction. A transaction must be committed for the changes to take effect.

Example

Listings 19.175 and 19.176 demonstrate how to create and commit a transaction.

Listing 19.175 Committing a Transaction (C#)

```
//Begin Transaction  
SqlTransaction nwTrans = nwConn.BeginTransaction("RemoveEmployees");  
  
//Delete Employee Territory  
SqlCommand nwCmd = new SqlCommand("", nwConn, nwTrans);  
nwCmd.CommandText = "DELETE FROM EmployeeTerritories WHERE EmployeeID=50";  
nwCmd.ExecuteNonQuery();  
  
nwTrans.Commit();
```

Listing 19.176 Committing a Transaction (Visual Basic.NET)

```
'Begin Transaction  
dim nwTrans as object  
nwTrans = nwConn.BeginTransaction("RemoveEmployees")  
  
'Delete Employee Territory  
dim nwCmd as new SqlCommand("", nwConn, nwTrans)  
nwCmd.CommandText = "DELETE FROM EmployeeTerritories WHERE EmployeeID=50"  
nwCmd.ExecuteNonQuery()  
  
nwTrans.Commit()
```

SqlTransaction.Rollback()

Syntax

```
Void Rollback()  
Void Rollback( String transactionName )
```

Description

The `RollBack()` method is used to roll back (that is, cancel) a transaction. If a string that contains the name of a saved point is passed as an argument, the transaction is rolled back to that save point.

Example

Listings 19.177 and 19.178 demonstrate how to roll back a transaction.

Listing 19.177 Rolling Back a Transaction (C#)

```
//Begin Transaction
SqlTransaction nwTrans = nwConn.BeginTransaction("RemoveEmployees");

//Delete Employee Territory
SqlCommand nwCmd = new SqlCommand("", nwConn, nwTrans);
nwCmd.CommandText = "DELETE FROM EmployeeTerritories WHERE EmployeeID=50";
nwCmd.ExecuteNonQuery();

//Rollback Transaction
nwTrans.Rollback();
```

Listing 19.178 Rolling Back a Transaction (Visual Basic.NET)

```
'Begin Transaction
dim nwTrans as object
nwTrans = nwConn.BeginTransaction("RemoveEmployees")

'Delete Employee Territory
dim nwCmd as new SqlCommand("", nwConn, nwTrans)
nwCmd.CommandText = "DELETE FROM EmployeeTerritories WHERE EmployeeID=50"
nwCmd.ExecuteNonQuery()

'Rollback Transaction
nwTrans.Rollback()
```

`SqlTransaction.Save()`

Syntax

```
Void Save()
```

Description

The `Save()` method enables a developer to bookmark a point in the transaction. The transaction can be rolled back to this saved point at any time. After the `SqlTransaction.Commit()` method has been called, you will be unable to roll back a transaction to a saved point.

Example

Listings 19.179 and 19.180 demonstrate how to save a database transaction at a particular point and then roll back the transaction to that point by using the `Rollback()` method.

Listing 19.179 Rolling Back a Transaction to a Saved Point (C#)

```
//Begin Transaction
SqlTransaction nwTrans = nwConn.BeginTransaction("RemoveEmployees");

//Delete Employee Territory
SqlCommand nwCmd = new SqlCommand("", nwConn, nwTrans);
nwCmd.CommandText = "DELETE FROM EmployeeTerritories WHERE EmployeeID=50";
nwCmd.ExecuteNonQuery();

//Save Transaction
nwTrans.Save("TerritoriesRemoved");

//Delete Another Employee
nwCmd.CommandText = "DELETE FROM Employees WHERE EmployeeID=60";
nwCmd.ExecuteNonQuery();

//Rollback Transaction to Save Point
nwTrans.Rollback("TerritoriesRemoved");
```

Listing 19.180 Rolling Back a Transaction to a Saved Point (Visual Basic.NET)

```
'Begin Transaction
dim nwTrans as object
nwTrans = nwConn.BeginTransaction("RemoveEmployees")

'Delete Employee Territory
dim nwCmd as new SqlCommand("", nwConn, nwTrans)
nwCmd.CommandText = "DELETE FROM EmployeeTerritories WHERE EmployeeID=50"
nwCmd.ExecuteNonQuery()

'Save Transaction
nwTrans.Save("TerritoriesRemoved")

'Delete Another Employee
nwCmd.CommandText = "DELETE FROM Employees WHERE EmployeeID=60"
nwCmd.ExecuteNonQuery()

'Rollback Transaction to Save Point
nwTrans.Rollback("TerritoriesRemoved")
```

CHAPTER 20

System.Web Reference

The `System.Web` namespace contains classes that simplify working with the browser, Hypertext Transfer Protocol (HTTP) requests, and HTTP responses. The `HttpResponse` class gives programmatic access to the HTTP output. Similarly, the `HttpRequest` class provides a wealth of information about an HTTP request. The `System.Web` namespace also contains classes that enable programmatic access to cookies and server variables.

The `HttpBrowserCapabilities` Class

The `HttpBrowserCapabilities` class is used to collect information about the browser that is requesting information from the Web server (see Table 20.1).

Table 20.1 Properties of the `HttpBrowserCapabilities` Class

Item	Description
ActiveXControls	Contains True if the browser supports ActiveX
AOL	Contains True if the browser is an AOL browser
BackgroundSounds	Contains True if the browser supports background sounds
Beta	Contains True if the browser is a beta version
Browser	Contains the browser string in the user-agent header
CDF	Contains True if the browser supports Channel Definition Format

Table 20.1 continued

Item	Description
Cookies	Contains True if the browser supports cookies
Crawler	Contains True if the program making the request is a Web crawler search engine
EcmaScriptVersion	Contains the version of ECMA script that the browser supports (ECMA is an international organization that is working to standardize information and communication systems)
Frames	Contains True if the browser supports frames
JavaApplets	Contains True if the browser supports Java applets
JavaScript	Contains True if the browser supports JavaScript
MajorVersion	Contains the major browser version
MinorVersion	Contains the minor browser version
MsDomVersion	Contains the version of the Microsoft XML document object model that the browser supports
Platform	Contains the name of the platform the client is using
Tables	Contains True if the browser supports tables
Type	Contains the name and major version number of the browser
VBScript	Contains True if the browser supports VBScript
Version	Contains the full browser version number (major plus minor)
W3CDomVersion	Contains the version of the W3C XML document object model that the browser supports
Win16	Contains True if the platform running the browser is 16 bit
Win32	Contains True if the platform running the browser is 32 bit

HttpBrowserCapabilities.ActiveXControls

Syntax

```
Boolean ActiveXControls
```

Description

The `ActiveXControls` property contains True if the browser supports Microsoft ActiveX controls. The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.1 and 20.2 demonstrate how to use the `ActiveXControls` property of the `HttpBrowserCapabilities` class.

Listing 20.1 Using the ActiveXControl Property (C#)

```
Boolean browCap = Request.Browser.ActiveXControls;
```

Listing 20.2 Using the ActiveXControl Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.ActiveXControls
```

HttpBrowserCapabilities.AOL

Syntax

```
Boolean AOL
```

Description

The AOL property contains True if the browser is an AOL browser. The information for the current browser request is contained in the Request.Browser object.

Example

Listings 20.3 and 20.4 demonstrate how to use the AOL property of the **HttpBrowserCapabilities** class.

Listing 20.3 Using the AOL Property (C#)

```
Boolean browCap = Request.Browser.AOL;
```

Listing 20.4 Using the AOL Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.AOL
```

HttpBrowserCapabilities.BackgroundSounds

Syntax

```
Boolean BackgroundSounds
```

Description

The **BackgroundSounds** property contains True if the browser supports background sounds. The information for the current browser request is contained in the Request.Browser object.

Example

Listings 20.5 and 20.6 demonstrate how to use the **BackgroundSounds** property of the **HttpBrowserCapabilities** class.

Listing 20.5 Using the BackgroundSounds Property (C#)

```
Boolean browCap = Request.Browser.BackgroundSounds;
```

Listing 20.6 Using the BackgroundSounds Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.BackgroundSounds
```

HttpBrowserCapabilities.Beta

Syntax

Boolean Beta

Description

The Beta property contains True if the browser is a beta version. The information for the current browser request is contained in the Request.Browser object.

Example

Listings 20.7 and 20.8 demonstrate how to use the Beta property of the HttpBrowserCapabilities class.

Listing 20.7 Using the Beta Property (C#)

```
Boolean browCap = Request.Browser.Beta;
```

Listing 20.8 Using the Beta Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.Beta
```

HttpBrowserCapabilities.Browser

Syntax

String Browser

Description

The Browser property contains the browser string in the user-agent header. The information for the current browser request is contained in the Request.Browser object.

Example

Listings 20.9 and 20.10 demonstrate how to use the Browser property of the HttpBrowserCapabilities class.

Listing 20.9 Using the Browser Property (C#)

```
string browCap = Request.Browser.Browser;
```

Listing 20.10 Using the Browser Property (Visual Basic.NET)

```
dim browCap as new string  
browCap = Request.Browser.Browser
```

HttpBrowserCapabilities.CDF

Syntax

Boolean CDF

Description

The `CDF` property contains `True` if the browser supports Channel Definition Format. The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.11 and 20.12 demonstrate how to use the `CDF` property of the `HttpBrowserCapabilities` class.

Listing 20.11 Using the `CDF` Property (C#)

```
Boolean browCap = Request.Browser.CDF;
```

Listing 20.12 Using the `CDF` Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.CDF
```

HttpBrowserCapabilities.Cookies

Syntax

```
Boolean Cookies
```

Description

The `Cookies` property contains `True` if the browser supports cookies. The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.13 and 20.14 demonstrate how to use the `Cookies` property of the `HttpBrowserCapabilities` class.

Listing 20.13 Using the `Cookies` Property (C#)

```
Boolean browCap = Request.Browser.Cookies;
```

Listing 20.14 Using the `Cookies` Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.Cookies
```

HttpBrowserCapabilities.Crawler

Syntax

```
Boolean Crawler
```

Description

The `Crawler` property contains `true` if the browser is a Web crawler search engine. The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.15 and 20.16 demonstrate how to use the `Crawler` property of the `HttpBrowserCapabilities` class.

Listing 20.15 Using the Crawler Property (C#)

```
Boolean browCap = Request.Browser.Crawler;
```

Listing 20.16 Using the Crawler Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.Crawler
```

HttpBrowserCapabilities.EcmaScriptVersion

Syntax

```
Version EcmaScriptVersion
```

Description

The `EcmaScriptVersion` property contains the version of ECMA script that the browser supports. The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.17 and 20.18 demonstrate how to use the `EcmaScriptVersion` property of the `HttpBrowserCapabilities` class.

Listing 20.17 Using the EcmaScriptVersion Property (C#)

```
string browCap = Request.Browser.EcmaScriptVersion.ToString();
```

Listing 20.18 Using the EcmaScriptVersion Property (Visual Basic.NET)

```
dim browCap as new string  
browCap = Request.Browser.EcmaScriptVersion.ToString()
```

HttpBrowserCapabilities.Frames

Syntax

```
Boolean Frames
```

Description

The `Frames` property contains `True` if the browser supports frames. The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.19 and 20.20 demonstrate how to use the `Frames` property of the `HttpBrowserCapabilities` class.

Listing 20.19 Using the `Frames` Property (C#)

```
Boolean browCap = Request.Browser.Frames;
```

Listing 20.20 Using the `Frames` Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.Frames
```

HttpBrowserCapabilities.JavaApplets

Syntax

```
Boolean JavaApplets
```

Description

The `JavaApplets` property contains `True` if the browser supports Java applets. The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.21 and 20.22 demonstrate how to use the `JavaApplets` property of the `HttpBrowserCapabilities` class.

Listing 20.21 Using the `JavaApplets` Property (C#)

```
Boolean browCap = Request.Browser.JavaApplets;
```

Listing 20.22 Using the `JavaApplets` Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.JavaApplets
```

HttpBrowserCapabilities.JavaScript

Syntax

```
Boolean JavaScript
```

Description

The `JavaScript` property contains `True` if the browser supports JavaScript. The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.23 and 20.24 demonstrate how to use the `JavaScript` property of the `HttpBrowserCapabilities` class.

Listing 20.23 Using the `JavaScript` Property (C#)

```
Boolean browCap = Request.Browser.JavaScript;
```

Listing 20.24 Using the JavaScript Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.JavaScript
```

HttpBrowserCapabilities.MajorVersion

Syntax

```
Int32 MajorVersion
```

Description

The `MajorVersion` property contains the major version number of the browser. (Note that the `MajorVersion` property returns an `Integer` data type and `MinorVersion` returns a `Double` data type.) The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.25 and 20.26 demonstrate how to use the `MajorVersion` property of the `HttpBrowserCapabilities` class.

Listing 20.25 Using the MajorVersion Property (C#)

```
int browCap = Request.Browser.MajorVersion;
```

Listing 20.26 Using the MajorVersion Property (Visual Basic.NET)

```
dim browCap as new Int32  
browCap = Request.Browser.MajorVersion
```

HttpBrowserCapabilities.MinorVersion

Syntax

```
Double MinorVersion
```

Description

The `MinorVersion` property contains the minor version number of the browser. (Note that the `MinorVersion` property returns a `Double` data type and `MajorVersion` returns an `Integer` data type.) The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.27 and 20.28 demonstrate how to use the `MinorVersion` property of the `HttpBrowserCapabilities` class.

Listing 20.27 Using the MinorVersion Property (C#)

```
Double browCap = Request.Browser.MinorVersion;
```

Listing 20.28 Using the MinorVersion Property (Visual Basic.NET)

```
dim browCap as new Int32  
browCap = Request.Browser.MinorVersion
```

HttpBrowserCapabilities.MsDomVersion

Syntax

```
Version MSDomVersion
```

Description

The `MsDomVersion` property contains the version of the Microsoft XML document object model that the browser supports. The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.29 and 20.30 demonstrate how to use the `MsDomVersion` property of the `HttpBrowserCapabilities` class.

Listing 20.29 Using the MsDomVersion Property (C#)

```
string browCap = Request.Browser.MsDomVersion.ToString();
```

Listing 20.30 Using the MsDomVersion Property (Visual Basic.NET)

```
dim browCap as string  
browCap = Request.Browser.MsDomVersion.ToString()
```

HttpBrowserCapabilities.Platform

Syntax

```
String Platform
```

Description

The `Platform` property contains the name of the platform in which the browser is running. The information for the current browser request is contained in the `Request.Browser` object.

Example

Listings 20.31 and 20.32 demonstrate how to use the `Platform` property of the `HttpBrowserCapabilities` class.

Listing 20.31 Using the Platform Property (C#)

```
string browCap = Request.Browser.Platform;
```

Listing 20.32 Using the Platform Property (Visual Basic.NET)

```
dim browCap as string  
browCap = Request.Browser.Platform
```

HttpBrowserCapabilities.Tables

Syntax

Boolean Tables

Description

The Tables property contains True if the browser supports tables. The information for the current browser request is contained in the Request.Browser object.

Example

Listings 20.33 and 20.34 demonstrate how to use the Tables property of the HttpBrowserCapabilities class.

Listing 20.33 Using the Tables Property (C#)

```
Boolean browCap = Request.Browser.Tables;
```

Listing 20.34 Using the Tables Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.Tables
```

HttpBrowserCapabilities.Type

Syntax

String Type

Description

The Type property contains the name and major version number of the browser. It is important to note that the minor version number of the browser is not returned. The HttpBrowserCapabilities.MinorVersion property can be used to access this information. The information for the current browser request is contained in the Request.Browser object.

Example

Listings 20.35 and 20.36 demonstrate how to use the Type property of the HttpBrowserCapabilities class.

Listing 20.35 Using the Type Property (C#)

```
string browCap = Request.Browser.Type;
```

Listing 20.36 Using the Type Property (Visual Basic.NET)

```
dim browCap as string  
browCap = Request.Browser.Type
```

HttpBrowserCapabilities.VBScript

Syntax

Boolean VBScript

Description

The VBScript property contains True if the browser supports VBScript. The information for the current browser request is contained in the Request.Browser object.

Example

Listings 20.37 and 20.38 demonstrate how to use the VBScript property of the **HttpBrowserCapabilities** class.

Listing 20.37 Using the VBScript Property (C#)

```
Boolean browCap = Request.Browser.VBScript;
```

Listing 20.38 Using the VBScript Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.VBScript
```

HttpBrowserCapabilities.Version

Syntax

String Version

Description

The Version property contains the full browser version number (major plus minor). The information for the current browser request is contained in the Request.Browser object.

Example

Listings 20.39 and 20.40 demonstrate how to use the Version property of the **HttpBrowserCapabilities** class.

Listing 20.39 Using the Version Property (C#)

```
string browCap = Request.Browser.Version;
```

Listing 20.40 Using the Version Property (Visual Basic.NET)

```
dim browCap as string  
browCap = Request.Browser.Version
```

HttpBrowserCapabilities.W3CDomVersion

Syntax

```
Version W3CDomVersion
```

Description

The W3CDomVersion property contains the version of the W3C XML document object model that the browser supports. The information for the current browser request is contained in the Request.Browser object.

Example

Listings 20.41 and 20.42 demonstrate how to use the W3CDomVersion property of the HttpBrowserCapabilities class.

Listing 20.41 Using the W3CDomVersion Property (C#)

```
string browCap = Request.Browser.W3CDomVersion.ToString();
```

Listing 20.42 Using the W3CDomVersion Property (Visual Basic.NET)

```
dim browCap as string  
browCap = Request.Browser.W3CDomVersion.ToString()
```

HttpBrowserCapabilities.Win16

Syntax

```
Boolean Win16
```

Description

The Win16 property contains True if the platform on which the browser runs is 16 bit. The information for the current browser request is contained in the Request.Browser object.

Example

Listings 20.43 and 20.44 demonstrate how to use the Win16 property of the HttpBrowserCapabilities class.

Listing 20.43 Using the Win16 Property (C#)

```
Boolean browCap = Request.Browser.Win16;
```

Listing 20.44 Using the Win16 Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.Win16
```

HttpBrowserCapabilities.Win32

Syntax

```
Boolean Win32
```

Description

The **Win32** property contains True if the platform on which the browser runs is 32 bit. The information for the current browser request is contained in the **Request.Browser** object.

Example

Listings 20.45 and 20.46 demonstrate how to use the **Win32** property of the **HttpBrowserCapabilities** class.

Listing 20.45 Using the Win32 Property (C#)

```
Boolean browCap = Request.Browser.Win32;
```

Listing 20.46 Using the Win32 Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.Win32
```

The **HttpCookie** Class

The **HttpCookie** class contains properties and methods necessary to work with individual cookies (see Table 20.2).

Table 20.2 Properties of the **HttpCookie Class**

no methods in	
HttpCookie objectItem	Description
Domain	Contains the domain of the cookie
Expires	Contains the expiration time of the cookie
HasKeys	Contains True if the cookie has subkeys
Name	Contains the name of the cookie
Path	Contains the virtual path to submit with the cookie
Secure	Contains True if the cookie is to be passed in a secure connection only
Value	Contains the value of the cookie
Values	Contains a collection of all cookie values

Listings 20.47 and 20.48 are examples of most of the properties in the **HttpCookie** class.

Listing 20.47 Working with Cookies (C#)

```
<% @Page Language="C#" %>
<%@ Import Namespace="System.Web" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {
            if (Page.IsPostBack) {

                //Create new cookie
                HttpCookie newCookie = new HttpCookie("UserName");

                //Configure cookie
                newCookie.Domain = "asppages.com";
                newCookie.Expires = new DateTime(2001, 12, 7);
                newCookie.Name = "UserName";
                newCookie.Path = "/";
                newCookie.Secure = false;
                newCookie.Value = txtUsername.Text;

                //Add cookie to response object
                Response.Cookies.Add(newCookie);

                //Output cookie value to page
                msg.Text = Response.Cookies["UserName"].Value;
            }
        }
    </script>

</HEAD>
<BODY>

<h1>Working With Cookies</h1>
<hr>

<form runat="server" id=form1 name=form1>

    Cookie contents: <asp:label id=msg runat=server></asp:label><br>
    <asp:TextBox id=txtUsername runat=server></asp:TextBox>
    <input type=submit runat=server>

</form>
```

Listing 20.47 continued

```
<hr>

</BODY>
</HTML>
```

Listing 20.48 Working with Cookies (Visual Basic.NET)

```
<% @Page Language="VB" %>
<%@ Import Namespace="System.Web" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >
        Sub Page_Load(Source as Object, E as EventArgs)
            If Page.IsPostBack Then

                'Create new cookie
                Dim newCookie As New HttpCookie("UserName")

                'Configure cookie
                newCookie.Domain = "asppages.com"
                newCookie.Expires = New DateTime(2001, 12, 7)
                newCookie.Name = "UserName"
                newCookie.Path = "/"
                newCookie.Secure = False
                newCookie.Value = txtUsername.Text

                'Add cookie to response object
                Response.Cookies.Add(newCookie)

                'Output cookie value to page
                msg.Text = Response.Cookies("UserName").Value
            End If
        End Sub
    </script>

</HEAD>
<BODY>

    <h1>Working With Cookies</h1>
    <hr>

    <form runat="server" id=form1 name=form1>
```

Listing 20.48 continued

```
Cookie contents: <asp:label id=msg runat=server></asp:label><br>
<asp:TextBox id=txtUsername runat=server></asp:TextBox>
<input type=submit runat=server id=submit1 name=submit1>

</form>
<hr>

</BODY>
</HTML>
```

HttpCookie.Domain

Syntax

```
String Domain
```

Description

The Domain property is a string containing the domain of the cookie.

Example

Listings 20.49 and 20.50 demonstrate how to set the Domain property of a cookie.

Listing 20.49 Setting the Domain Property of a Cookie (C#)

```
Response.Cookies["MyNewCookie"].Domain = "asppages.com";
```

Listing 20.50 Setting the Domain Property of a Cookie (Visual Basic.NET)

```
Response.Cookies("MyNewCookie").Domain = "asppages.com"
```

HttpCookie.Expires

Syntax

```
DateTime Expires
```

Description

The Expires property is used to specify a date at which a cookie becomes invalid.

Example

Listings 20.51 and 20.52 demonstrate how to set the expiration date of a cookie.

Listing 20.51 Specifying the Expiration Date of a Cookie (C#)

```
DateTime toExpire = new DateTime(2001, 12, 7);
Response.Cookies["MyNewCookie"].Expires = toExpire;
```

Listing 20.52 Specifying the Expiration Date of a Cookie (Visual Basic.NET)

```
dim toExpire as new DateTime(2001, 12, 7)
Response.Cookies("MyNewCookie").Expires = toExpire
```

HttpCookie.HasKeys

Syntax

```
Boolean HasKeys
```

Description

The `HasKeys` property is True if the cookie has subkeys.

Example

Listings 20.53 and 20.54 demonstrate how to use the `HasKeys` property.

Listing 20.53 Using the HasKeys Property (C#)

```
Request.Cookies["MyNewCookie"].HasKeys;
```

Listing 20.54 Using the HasKeys Property (Visual Basic.NET)

```
Request.Cookies("MyNewCookie").HasKeys
```

HttpCookie.Name

Syntax

```
String Name
```

Description

The `Name` property contains the name of the cookie.

Example

Listings 20.55 and 20.56 demonstrate how to retrieve the name of a cookie.

Listing 20.55 Using the commandText Property (C#)

```
string cookieName = Request.Cookies["MyNewCookie"].Name;
```

Listing 20.56 Using the commandText Property (Visual Basic.NET)

```
dim cookieName as string
CookieName = Request.Cookies("MyNewCookie").Name
```

HttpCookie.Path

Syntax

```
String Path
```

Description

The Path property contains the virtual path associated with the current cookie.

Example

Listings 20.57 and 20.58 demonstrate how to retrieve the virtual path associated with a cookie.

Listing 20.57 Retrieving the Path of a Cookie (C#)

```
string cookiePath = Request.Cookies["MyNewCookie"].Path;
```

Listing 20.58 Retrieving the Path of a Cookie (Visual Basic.NET)

```
dim cookiePath as string  
cookiePath = Request.Cookies("MyNewCookie").Path
```

HttpCookie.Secure

Syntax

```
Boolean Secure
```

Description

The Secure property is True if the cookie should be sent only over a secure connection.

Example

Listings 20.59 and 20.60 demonstrate how to retrieve the Secure property of a cookie.

Listing 20.59 Retrieving the Secure Property (C#)

```
Boolean transmitSecure = Request.Cookies["MyNewCookie"].Secure;
```

Listing 20.60 Retrieving the secure Property (Visual Basic.NET)

```
dim transmitSecure as Boolean  
transmitSecure = Request.Cookies("MyNewCookie").Secure
```

HttpCookie.Value

Syntax

```
String Value
```

Description

The Value property contains the actual text value of the cookie.

Example

Listings 20.61 and 20.62 demonstrate how to set the value of a cookie.

Listing 20.61 Setting the Value of a Cookie (C#)

```
Response.Cookies["MyNewCookie"].Value = "Test Value";
```

Listing 20.62 Setting the Value of a Cookie (Visual Basic.NET)

```
Response.Cookies("MyNewCookie").Value = "Test Value"
```

HttpCookie.Values

Syntax

```
NameValueCollection Values
```

Description

The `Values` property is a collection of names–value pairs that enable a cookie to contain subvalues.

Example

Listings 20.63 and 20.64 demonstrate how to use the `Values` collection to set and retrieve a few subkeys for a cookie.

Listing 20.63 Specifying Multiple Values in the Same Cookie (C#)

```
Response.Cookies["MyNewCookie"].Values["SubCookie1"] = "value1";
Response.Cookies["MyNewCookie"].Values["SubCookie2"] = "value2";
```

```
string myResponse = Request.Cookies["MyNewCookie"].Values["SubCookie1"] + " ";
myResponse += Request.Cookies["MyNewCookie"].Values["SubCookie2"];
```

Listing 20.64 Specifying Multiple Values in the Same Cookie (Visual Basic.NET)

```
Response.Cookies("MyNewCookie").Values("SubCookie1") = "value1"
Response.Cookies("MyNewCookie").Values("SubCookie2") = "value2"
```

```
dim myResponse as String = Request.Cookies("MyNewCookie").Values("SubCookie1")
+ " "
myResponse = myResponse + Request.Cookies("MyNewCookie").Values("SubCookie2")
```

The `HttpRequest` Class

The `HttpRequest` class contains properties and methods necessary to handle an HTTP request (see Table 20.3). It contains all information passed by the browser, including all form variables, certificates and header information. It also contains the CGI server variables.

Table 20.3 Properties and Methods of the `HttpRequest` Class

Item	Description
Properties	
AcceptTypes	Contains a list of supported multipurpose Internet mail extensions (MIME) accept types
ApplicationPath	Contains the application root path
Browser	Contains information about the browser that is performing the request
ClientCertificate	Contains the client security certificate for the current request
ContentEncoding	Contains the character set of the current request
ContentLength	Contains the length of the content sent by the browser for the current request
ContentType	Contains the MIME content type for the current request
Cookies	Contains the collection of cookies sent by the browser
FilePath	Contains the virtual path of the application
Files	Contains the names of the files uploaded by the browser
Form	Contains the collection of form variables
Headers	Contains the collection of HTTP headers
HttpMethod	Contains the HTTP data transfer method
IsAuthenticated	Contains True if the user has been authenticated
IsSecureConnection	Contains True if the current request was made over a secure connection
Params	Contains a combined collection of <code>QueryString</code> , <code>Form</code> , <code>ServerVariable</code> , and <code>Cookie</code> collections
Path	Contains the virtual path for the current request
PathInfo	Contains additional path information
PhysicalApplicationPath	Contains the physical disk location of the root directory for the application
PhysicalPath	Contains the physical disk location of the requested uniform resource locator (URL)
QueryString	Contains a collection of querystring values for the current request
RawUrl	Contains the complete, unmodified URL for the current request
RequestType	Contains the data transfer method for the current request
ServerVariables	Contains a collection of Web server variables
TotalBytes	Contains the total number of bytes of the current request
Url	Contains information about the URL of the current request

Table 20.3 continued

Item	Description
Properties	
<code>UrlReferrer</code>	Contains information about the previous request
<code>UserAgent</code>	Contains the contents of the user-agent string, as sent by the browser
<code>UserHostAddress</code>	Contains the remote Internet Protocol (IP) address of the client
<code>UserHostName</code>	Contains the remote host name of the client
<code>UserLanguages</code>	Contains a string array of client language preferences
Methods	
<code>MapPath()</code>	Maps a virtual path to the actual physical path for the current request
<code>SaveAs()</code>	Saves the entire HTTP request to disk

HttpRequest.AcceptTypes

Syntax

```
String[] AcceptTypes
```

Description

The `AcceptTypes` property contains an array of all accepted MIME types.

Example

Listings 20.65 and 20.66 demonstrate how to list all browser-accepted MIME types.

Listing 20.65 Using the `AcceptTypes` Property (C#)

```
int index;
string output;

String[] myArray = Request.AcceptTypes;
for ( index = 0; index < myArray.Length; index++ ) {
    output = "Accept Type " + index + ": " + myArray[index] + "<br>";
}
```

Listing 20.66 Using the `AcceptTypes` Property (Visual Basic.NET)

```
Dim MyArray() As String
Dim output as String
Dim index As Integer

MyType = Request.AcceptTypes
For index = 0 To Ubound(MyArray)
    output = "Accept Type " + index + ": " + myArray(index) + "<br>"
Next index
```

HttpRequest.ApplicationPath

Syntax

```
String ApplicationPath
```

Description

The ApplicationPath property contains the virtual path of the application.

Example

Listings 20.67 and 20.68 demonstrate how to retrieve the virtual path.

Listing 20.67 Using the ApplicationPath Property (C#)

```
string AppPath = Request.ApplicationPath;
```

Listing 20.68 Using the ApplicationPath Property (Visual Basic.NET)

```
dim AppPath as string  
AppPath = Request.ApplicationPath
```

HttpRequest.Browser

Syntax

```
HttpBrowserCapabilities Browser
```

Description

The Browser object contains information about the capabilities of the browser that is making the request. More information about this object can be found in the [Browser class section of the System.Web namespace reference](#).

Example

Listings 20.69 and 20.70 demonstrate how to retrieve the browser object.

Listing 20.69 Retrieving the Browser Object (C#)

```
HttpBrowserCapabilities BrowCaps = new HttpBrowserCapabilities();  
BrowCaps = Request.Browser;
```

Listing 20.70 Retrieving the Browser Object (Visual Basic.NET)

```
dim BrowCaps as new HttpBrowserCapabilities()  
BrowCaps = Request.Browser
```

HttpRequest.ClientCertificate

Syntax

```
HttpClientCertificate ClientCertificate
```

Description

The `ClientCertificate` property returns an object containing information about the client's security certificate.

Example

Listings 20.71 and 20.72 demonstrate how to retrieve the `ClientCertificate` object for the current client.

Listing 20.71 Using the `clientCertificate` Property (C#)

```
Object clientCertificate = Request.ClientCertificate;
```

Listing 20.72 Using the `clientCertificate` Property (Visual Basic.NET)

```
dim clientCertificate as Object  
clientCertificate = Request.ClientCertificate
```

`HttpRequest.ContentEncoding`

Syntax

```
Encoding ContentEncoding
```

Description

The `ContentEncoding` property contains an object with information about the character set of the client.

Example

Listings 20.73 and 20.74 demonstrate how to retrieve the `EncodingName` property of the `ContentEncoding` object.

Listing 20.73 Using the `contentEncoding` Property (C#)

```
String EncodingType;  
EncodingType = Request.ContentEncoding.EncodingName;
```

Listing 20.74 Using the `contentEncoding` Property (Visual Basic.NET)

```
Dim EncodingType As String  
EncodingType = Request.ContentEncoding.EncodingName
```

`HttpRequest.ContentLength`

Syntax

```
String ContentLength
```

Description

The `ContentLength` property contains the length of the entire client request.

Example

Listings 20.75 and 20.76 demonstrate how to retrieve the length of the client request.

Listing 20.75 Using the ContentLength Property (C#)

```
String contentLength = Request.ContentLength;
```

Listing 20.76 Using the ContentLength Property (Visual Basic.NET)

```
dim contentLength as string  
contentLength = Request.ContentLength
```

HttpRequest.ContentType

Syntax

```
String ContentType
```

Description

The ContentType property contains information about the MIME content type of the client making the request.

Example

Listings 20.77 and 20.78 demonstrate how to retrieve the value of the ContentType property of the current request.

Listing 20.77 Using the ContentType Property (C#)

```
String contentType = Request.ContentType;
```

Listing 20.78 Using the ContentType Property (Visual Basic.NET)

```
dim contentType as string  
contentType = Request.ContentType
```

HttpRequest.Cookies

Syntax

```
HttpCookieCollection Cookies
```

Description

The Cookies property contains a collection of all the cookies passed by the client in the current request.

Example

Listings 20.79 and 20.80 demonstrate how to retrieve the cookie collection for the current request. More information on cookies can be found in the `Cookie` class in the `System.Web` reference.

Listing 20.79 Using the cookie Property (C#)

```
HttpCookieCollection cookies = Request.Cookies;
```

Listing 20.80 Using the cookie Property (Visual Basic.NET)

```
dim cookies as new HttpCookieCollection()  
cookies = Request.Cookies
```

HttpRequest.FilePath

Syntax

```
String FilePath
```

Description

The `FilePath` property contains the virtual path of the current request. The information returned does not contain any querystring information appended to the end of the URL.

Example

Listings 20.81 and 20.82 demonstrate how to retrieve the value of the `FilePath` property for a request.

Listing 20.81 Using the FilePath Property (C#)

```
string filePath = Request.FilePath;
```

Listing 20.82 Using the FilePath Property (Visual Basic.NET)

```
dim filePath as string  
filePath = Request.FilePath
```

HttpRequest.Files

Syntax

```
HttpFileCollection Files
```

Description

The `Files` property contains a collection of files uploaded by the client in the current request. This collection is populated only if the client uploads files.

Example

Listings 20.83 and 20.84 demonstrate how to use a form to upload a file from the client and then gather information about the file and save it to the requested location on the server.

Listing 20.83 Using the Files Property (C#)

```
<% @Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="C#" runat="server" >
        void Page_Load(Object Source, EventArgs E)
        {
            if(IsPostBack==true) {

                string[] fileArray = new string[Request.Files.Count];
                fileArray = Request.Files.AllKeys;

                //For simplicity of presentation, only look at first file
                fileName.Text = Request.Files.Get(0).FileName;
                fileSize.Text = Request.Files.Get(0).ContentLength.ToString();
                fileType.Text = Request.Files.Get(0).ContentType;

                try {
                    Request.Files.Get(0).SaveAs(txtSaveLocation.Text);
                }
                catch( Exception e) {
                }
            }
        }
    </script>

</HEAD>
<BODY>

<h1>Uploading Files</h1>
<hr>

<form enctype="multipart/form-data" runat="server">
    <p>
        File To Upload: <br>
        <input type="file" id="fileUpload" runat=server>
    </p>

    <p>
        Location to Save on Server (including file name): <br>
        <asp:TextBox id="txtSaveLocation" runat="server"></asp:TextBox><br>
    </p>
</form>
```

Listing 20.83 continued

```

<input type="submit">
</p>
<p>
<h2>File Information:</h2>
<b>File Name:</b> <asp:label id=fileName runat="server"></asp:label><br>
<b>File Size:</b> <asp:label id=FileSize runat="server"></asp:label><br>
<b>File Type:</b> <asp:label id=FileType runat="server"></asp:label><br>
</p>
</form>
<hr>

</BODY>
</HTML>

```

Listing 20.84 Using the `Files` Property (Visual Basic.NET)

```

<% @Page Language="VB" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<HTML>
<HEAD>
    <LINK rel="stylesheet" type="text/css" href="Main.css">
    <!-- End Style Sheet -->

    <script language="VB" runat="server" >
        Sub Page_Load(Source as object, E as EventArgs)

            if IsPostBack=true then
                dim fileArray(Request.Files.Count) as string
                fileArray = Request.Files.AllKeys

                'string[] fileArray = new string[Request.Files.Count];
                'fileArray = Files.AllKeys;

                'For simplicity of presentation, only look at first file
                fileName.Text = Request.Files.Get(0).FileName
                fileSize.Text = Request.Files.Get(0).ContentLength.ToString()
                fileType.Text = Request.Files.Get(0).ContentType

            Try
                Request.Files.Get(0).SaveAs(txtSaveLocation.Text)
            Catch
                'Do nothing
            End Try
            end if

        End Sub
    
```

Listing 20.84 continued

```
</script>

</HEAD>
<BODY>

<h1>Uploading Files</h1>
<hr>

<form enctype="multipart/form-data" runat="server" id=form1 name=form1>
<p>
File To Upload: <br>
<input type="file" id="fileUpload" runat=server>
</p>

<p>
Location to Save on Server (including file name): <br>
<asp:TextBox id="txtSaveLocation" runat="server"></asp:TextBox><br>

<input type="submit" id=submit1 name=submit1>
</p>
<p>
<h2>File Information:</h2>
<b>File Name:</b> <asp:label id=fileName runat="server"></asp:label><br>
<b>File Size:</b> <asp:label id=fileSize runat="server"></asp:label><br>
<b>File Type:</b> <asp:label id=fileType runat="server"></asp:label><br>
</p>
</form>
<hr>

</BODY>
</HTML>
```

HttpRequest.Form

Syntax

NameValueCollection Form

Description

The Form property is a collection of all form values passed from the client during the submission of a form.

Example

Listings 20.85 and 20.86 demonstrate how to retrieve a value from the forms collection. The examples assume that the client just submitted a form with a field named txtName.

Listing 20.85 Using the Form Property (C#)

```
string formValue = Request.Form["txtName"];
```

Listing 20.86 Using the Form Property (Visual Basic.NET)

```
dim formValue as string  
formValue = Request.Form("txtName")
```

HttpRequest.Headers

Syntax

```
NameValueCollection Headers
```

Description

The `Headers` property is a collection of all the headers passed by the client in the request.

Example

Listings 20.87 and 20.88 demonstrate how to use the headers collection to retrieve the content-length header.

Listing 20.87 Using the Headers Property (C#)

```
string contentLength = Request.Headers["Content-Length"];
```

Listing 20.88 Using the Headers Property (Visual Basic.NET)

```
dim contentLength as string  
contentLength = Request.Headers("Content-Length")
```

HttpRequest.HttpMethod

Syntax

```
String HttpMethod
```

Description

The `HttpMethod` property is used to determine the type of data transfer method used by the form. The three possible values are `GET`, `POST`, and `HEAD`.

Example

Listings 20.89 and 20.90 demonstrate how to retrieve the value of the `HttpMethod` property of a form.

Listing 20.89 Using the HttpMethod Property (C#)

```
string RequestMethod = Request.HttpMethod;
```

Listing 20.90 Using the HttpMethod Property (Visual Basic.NET)

```
dim requestMethod as string  
requestMethod = Request.HttpMethod
```

HttpRequest.IsAuthenticated

Syntax

```
Boolean IsAuthenticated
```

Description

The `IsAuthenticated` property contains `True` if the remote user has been authenticated.

Example

Listings 20.91 and 20.92 demonstrate how to retrieve the value of the `IsAuthenticated` property.

Listing 20.91 Using the IsAuthenticated Property (C#)

```
Boolean isAuthenticated = Request.IsAuthenticated;
```

Listing 20.92 Using the IsAuthenticated Property (Visual Basic.NET)

```
dim isAuthenticated as Boolean  
isAuthenticated = Request.IsAuthenticated
```

HttpRequest.IsSecureConnection

Syntax

```
Boolean IsSecureConnection
```

Description

The `IsSecureConnection` property contains `True` if the current request was made over a secure connection (that is, the URL starts with `https://`).

Example

Listings 20.93 and 20.94 demonstrate how to retrieve the value of the `IsSecureConnection` property.

Listing 20.93 Using the IsSecureConnection Property (C#)

```
Boolean IsSecureConnection = Request.IsSecureConnection;
```

Listing 20.94 Using the IsSecureConnection Property (Visual Basic.NET)

```
dim isSecureConnection as Boolean  
IsSecureConnection = Request.IsSecureConnection
```

HttpRequest.Params

Syntax

```
NameValueCollection Params
```

Description

The `Params` property contains a combined collection of all querystring, form, cookie, and server variables submitted with the current request.

Example

Listings 20.95 and 20.96 use the `Params` property to build a string containing all the values present.

Listing 20.95 Using the Params Property (C#)

```
string output;  
  
for( int i=0; i < Request.Params.Count; i++)  
    output += Request.Params[i] + "<BR>";
```

Listing 20.96 Using the Params Property (Visual Basic.NET)

```
dim i as Int32  
dim output as string  
  
for i=0 to i < Request.Params.Count  
    output = output + Request.Params(i) + "<BR>"  
next i
```

HttpRequest.Path

Syntax

```
String Path
```

Description

The `Path` property contains the virtual path for the current request.

Example

Listings 20.97 and 20.98 demonstrate how to retrieve the virtual path of the current request.

Listing 20.97 Using the Path Property (C#)

```
string virtualPath = Request.Path;
```

Listing 20.98 Using the Path Property (Visual Basic.NET)

```
dim virtualPath as string  
virtualPath = Request.Path
```

HttpRequest.PathInfo

Syntax

String PathInfo

Description

The PathInfo property contains additional path information for the current request. Specifically, the PathInfo property is empty unless the requested URL has a tail with appended information, in the form `http://www.asppages.com/default.asp/tail`.

Example

Listings 20.99 and 20.100 demonstrate how to retrieve the PathInfo property for the current request.

Listing 20.99 Using the PathInfo Property (C#)

```
string pathInfo = Request.PathInfo;
```

Listing 20.100 Using the PathInfo Property (Visual Basic.NET)

```
dim pathInfo as string  
pathInfo = Request.PathInfo
```

HttpRequest.PhysicalApplicationPath

Syntax

String PhysicalApplicationPath

Description

The PhysicalApplicationPath property contains the physical file system directory of the current application.

Example

Listings 20.101 and 20.102 demonstrate how to retrieve the physical application path for the current request.

Listing 20.101 Using the PhysicalApplicationPath Property (C#)

```
string appPath = Request.PhysicalApplicationPath;
```

Listing 20.102 Using the PhysicalApplicationPath Property (Visual Basic.NET)

```
dim appPath as string  
appPath = Request.PhysicalApplicationPath
```

HttpRequest.PhysicalPath

Syntax

```
String PhysicalPath
```

Description

The `PhysicalPath` property contains the physical file system path of the current request.

Example

Listings 20.103 and 20.104 demonstrate how to retrieve the physical path of the current request.

Listing 20.103 Using the PhysicalPath Property (C#)

```
string physicalPath = Request.PhysicalPath;
```

Listing 20.104 Using the PhysicalPath Property (Visual Basic.NET)

```
dim physicalPath as string  
physicalPath = Request.PhysicalPath
```

HttpRequest.QueryString

Syntax

```
NameValueCollection QueryString
```

Description

The `QueryString` property is a collection of all querystring items in the current request.

Example

Listings 20.105 and 20.106 demonstrate how to retrieve the querystring from the following URL: <http://www.nerdgod.com/foo.asp?txtName=bull>.

Listing 20.105 Using the QueryString Property (C#)

```
string myName = Request.QueryString["txtName"];
```

Listing 20.106 Using the QueryString Property (Visual Basic.NET)

```
dim myName as string  
myName = Request.QueryString("txtName")
```

HttpRequest.RawUrl

Syntax

```
String RawUrl
```

Description

The RawUrl property contains the entire URL for the current request. If the URL is `http://www.asppages.com/default.aspx?auth=true`, then the RawUrl property is `/default.aspx?auth=true`.

Example

Listings 20.107 and 20.108 demonstrate how to retrieve the value of the RawUrl property for the current request.

Listing 20.107 Using the RawUrl Property (C#)

```
string rawURL = Request.RawUrl;
```

Listing 20.108 Using the RawUrl Property (Visual Basic.NET)

```
dim rawURL as string  
rawURL = Request.RawUrl
```

HttpRequest.RequestType

Syntax

```
String RequestType
```

Description

The RequestType property contains the data transfer method for the current request. The value contains GET or POST.

Example

Listings 20.109 and 20.110 demonstrate how to retrieve the value of the RequestType property for the current request.

Listing 20.109 Using the RequestType Property (C#)

```
string requestType = Request.RequestType;
```

Listing 20.110 Using the RequestType Property (Visual Basic.NET)

```
dim requestType as string  
requestType = Request.RequestType
```

HttpRequest.ServerVariables

Syntax

```
NameValueCollection ServerVariables
```

Description

The ServerVariables property contains a collection of all the Web server variables for the current request.

Example

Listings 20.111 and 20.112 use the `ServerVariables` property to build a string containing all server variables.

Listing 20.111 Using the ServerVariables Property (C#)

```
string output = "";

for( int i=0; i < Request.ServerVariables.Count; i++)
    output += Request.ServerVariables[i] + "<BR>";
```

Listing 20.112 Using the ServerVariables Property (Visual Basic.NET)

```
dim output as string
dim i as Int32

for i=0 to i < Request.ServerVariables.Count
    output = output + Request.ServerVariables(i) + "<BR>"
next i
```

HttpRequest.TotalBytes

Syntax

```
Int32 TotalBytes
```

Description

The `TotalBytes` property contains the size of the current request, in bytes.

Example

Listings 20.113 and 20.114 demonstrate how to get the size of the current request.

Listing 20.113 Using the TotalBytes Property (C#)

```
int totalBytes = Request.TotalBytes;
```

Listing 20.114 Using the TotalBytes Property (Visual Basic.NET)

```
dim totalBytes as Int32
totalBytes = Request.TotalBytes
```

HttpRequest.Url

Syntax

```
Uri Url
```

Description

The `Url` property returns an object of type `Uri` (defined in the `System` namespace).

Example

Listings 20.115 and 20.116 demonstrate how to retrieve the absolute path of a URL by accessing the `AbsolutePath` property.

Listing 20.115 Using the Uri Property (C#)

```
Uri UrlInfo = Request.Url;  
string absolutePath = UrlInfo.AbsolutePath;
```

Listing 20.116 Using the Uri Property (Visual Basic.NET)

```
dim UrlInfo as Uri  
dim absolutePath as string  
  
UrlInfo = Request.Url  
absolutePath = UrlInfo.AbsolutePath
```

HttpRequest.UrlReferrer

Syntax

```
Uri UrlReferrer
```

Description

The `UrlReferrer` property contains information about the referring URL (that is, the page the user was visiting before accessing the current page).

Example

Listings 20.117 and 20.118 demonstrate how to retrieve the absolute path of the linking page.

Listing 20.117 Using the UrlReferrer Property (C#)

```
Uri UrlReferrerInfo = Request.Url;  
string absolutePath = UrlReferrerInfo.AbsolutePath;
```

Listing 20.118 Using the UrlReferrer Property (Visual Basic.NET)

```
dim UrlReferrerInfo as Uri  
dim absolutePath as string  
  
UrlReferrerInfo = Request.UrlReferrer  
absolutePath = UrlReferrerInfo.AbsolutePath
```

HttpRequest.UserAgent

Syntax

```
String UserAgent
```

Description

The `UserAgent` property contains the entire user-agent header for the browser that is making the request. For the version of Microsoft Internet Explorer that ships with Windows 2000, the user-agent is Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0; COM+ 1.0.2615).

Example

Listings 20.119 and 20.120 demonstrate how to retrieve the user-agent header for the current request.

Listing 20.119 Using the `UserAgent` Property (C#)

```
string userAgent = Request.UserAgent;
```

Listing 20.120 Using the `UserAgent` Property (Visual Basic.NET)

```
dim userAgent as string  
userAgent = Request.UserAgent
```

HttpRequest.UserHostAddress

Syntax

```
String UserHostAddress
```

Description

The `UserHostAddress` property contains the IP address of the remote client that is making the request.

Example

Listings 20.121 and 20.122 demonstrate how to retrieve the remote IP address of a client.

Listing 20.121 Using the `UserHostAddress` Property (C#)

```
string remoteAddr = Request.UserHostAddress;
```

Listing 20.122 Using the `UserHostAddress` Property (Visual Basic.NET)

```
dim remoteAddr as string  
remoteAddr = Request.UserHostAddress
```

HttpRequest.UserHostName

Syntax

```
String UserHostName
```

Description

The UserHostName property contains the remote Domain Name Server (DNS) host name of the client that is performing the current request.

Example

Listings 20.123 and 20.124 demonstrate how to retrieve the remote DNS host name of the client that is performing the request.

Listing 20.123 Using the UserHostName Property (C#)

```
string remoteHost = Request.UserHostName
```

Listing 20.124 Using the UserHostName Property (Visual Basic.NET)

```
dim remoteHost as string  
remoteHost = Request.UserHostName
```

HttpRequest.UserLanguages

Syntax

```
String[] UserLanguages
```

Description

The UserLanguages property contains a string array of the language preferences of the client.

Example

Listings 20.125 and 20.126 demonstrate how to retrieve all the language preferences for the current client.

Listing 20.125 Using the UserLanguages Property (C#)

```
string output = "";  
  
for( int i=0; i < Request.UserLanguages.Length; i++ )  
    output = output + Request.UserLanguages[i] + "<BR>;
```

Listing 20.126 Using The UserLanguages Property (Visual Basic.NET)

```
dim output as string  
dim i as Int32  
  
for i=0 to i < Request.UserLanguages.Count  
    output = output + Request.UserLanguages(i) + "<BR>"  
next i
```

`HttpRequest.MapPath()`

Syntax

```
String MapPath( String virtualPath )
String MapPath( String virtualPath, String baseVirtualDir,
                Boolean allowCrossAppMapping )
```

Description

The `MapPath()` method accepts a string containing a virtual path and returns the actual physical application directory for the virtual path.

Example

Listings 20.127 and 20.128 demonstrate how to use the `MapPath()` method to find the root Internet Information Server (IIS) directory on a server.

Listing 20.127 Using the `MapPath()` Method (C#)

```
string rootIisPath = MapPath("//");
```

Listing 20.128 Using the `MapPath()` Method (Visual Basic.NET)

```
dim rootIisPath as string
rootIisPath = MapPath("//")
```

`HttpRequest.SaveAs()`

Syntax

```
Void SaveAs( String filename, Boolean includeHeaders )
```

Description

The `SaveAs()` method saves the entire request to disk. The first argument is a string containing a physical disk location. If the second argument is true, then the headers will be saved to disk as well. The `SaveAs()` method could be used to save a client request to disk in the case of an error so that it could be fully debugged later.

Example

Listings 20.129 and 20.130 demonstrate how to save the current request to disk.

Listing 20.129 Using the `SaveAs()` Method (C#)

```
Request.SaveAs("c:\\temp\\entireRequest.txt", true);
```

Listing 20.130 Using the `SaveAs()` Method (Visual Basic.NET)

```
Request.SaveAs("c:\\temp\\entireRequest.txt", true)
```

The `HttpResponse` Class

The `HttpResponse` class contains methods and properties necessary to crafting an HTTP response (see Table 20.4). Several methods are used to modify headers, cookies, and the actual content of the response.

Table 20.4 Properties and Methods of the `HttpResponse` Class

Item	Description
Properties	
Buffer	If set to True, the entire response is buffered in memory until the response is finished before sending to client
BufferOutput	If set to True, the entire response is buffered in memory until the response is finished before sending to client
CacheControl	Sets the response cache-control header to Public or Private
Charset	Contains the HTTP character set for the response
ContentEncoding	Contains the HTTP character set for the response
ContentType	Contains the HTTP MIME type of the response
Cookies	Contains the response cookie collection (that is, the cookies that will be transmitted back to the client on the current response)
Expires	Contains the number of minutes before the page transmitted to the client expires
ExpiresAbsolute	Contains the absolute date and time for a page to be removed from the cache
IsClientConnected	Contains True if the client is still connected to the Web server
Status	Contains the HTTP status sent to the client
StatusCode	Contains the integer value of the HTTP status sent to the client
StatusDescription	Contains the HTTP status description sent to the client
SuppressContent	Contains True if HTML content should not be sent to the client
Methods	
AddFileDependency()	Adds a single file to the collection of files that the current response depends on
AddHeader()	Adds another header to the header collection
AppendCookie()	Adds a cookie to the cookie collection
AppendHeader()	Adds another header to the header collection
AppendToLog()	Adds log information to the IIS log file
BinaryWrite()	Writes a string of binary characters to the output stream
Clear()	Clears the buffer stream for the current response
ClearContent()	Clears the buffer stream for the current response
ClearHeaders()	Clears all headers from the output stream

Table 20.4 continued

Item	Description
Methods	
Close()	Closes the connection to the client
End()	Sends all buffered output to the client and ceases execution of the current response
Flush()	Sends all buffered output to the client
Pics()	Adds a Platform for Internet Content Selection (PICS)-Label header to the header collection
Redirect()	Redirects a client to a new URL
Write()	Writes information to the output stream
WriteFile()	Writes a file directly to the output stream

HttpResponse.Buffer

Syntax

```
Boolean Buffer
```

Description

The Buffer property enables a developer to specify that the entire response should be held in memory until processing is finished and then sent to the client.

Example

Listings 20.131 and 20.132 demonstrate how to set the Buffer property.

Listing 20.131 Using the Buffer Property (C#)

```
Response.Buffer = true;
```

Listing 20.132 Using the Buffer Property (Visual Basic.NET)

```
Response.Buffer = true
```

HttpResponse.BufferOutput

Syntax

```
Boolean BufferOutput
```

Description

The BufferOutput property enables a developer to specify that the entire response should be held in memory until processing is finished and then sent to the client.

Example

Listings 20.133 and 20.134 demonstrate how to buffer the contents of a response until processing is finished.

Listing 20.133 Using the BufferOutput Property (C#)

```
Response.BufferOutput = true;
```

Listing 20.134 Using the BufferOutput Property (Visual Basic.NET)

```
Response.BufferOutput = true;
```

HttpResponse.CacheControl

Syntax

```
String CacheControl
```

Description

The CacheControl property is used to set the cache-control HTTP header to Public or Private.

Example

Listings 20.135 and 20.136 demonstrate how to set the cache-control HTTP header to Public.

Listing 20.135 Using the CacheControl Property (C#)

```
Response.CacheControl = "Public";
```

Listing 20.136 Using the CacheControl Property (Visual Basic.NET)

```
Response.CacheControl = "Public"
```

HttpResponse.Charset

Syntax

```
String Charset
```

Description

The Charset property contains the charset of the response.

Example

Listings 20.137 and 20.138 demonstrate how to retrieve the character set (charset) of the current response.

Listing 20.137 Using the Charset Property (C#)

```
string charSet = Response.Charset;
```

Listing 20.138 Using the Charset Property (Visual Basic.NET)

```
dim charSet as string  
charSet = Response.Charset
```

HttpResponse.ContentEncoding

Syntax

```
Encoding ContentEncoding
```

Description

The `ContentEncoding` property is an encoding object that contains information about the character set of the current response.

Example

Listings 20.139 and 20.140 demonstrate how to retrieve the friendly name of the character set for the current response.

Listing 20.139 Using the ContentEncoding Property (C#)

```
string charSetName = Response.ContentEncoding.EncodingName;
```

Listing 20.140 Using the ContentEncoding Property (Visual Basic.NET)

```
dim charSetName as string  
charSetName = Response.ContentEncoding.EncodingName
```

HttpResponse.ContentType

Syntax

```
String ContentType
```

Description

The `ContentType` property contains the HTTP MIME type of the current response.

Example

Listings 20.141 and 20.142 demonstrate how to retrieve the HTTP MIME type of the current response.

Listing 20.141 Using the ContentType Property (C#)

```
string contentType = Response.ContentType;
```

Listing 20.142 Using the ContentType Property (Visual Basic.NET)

```
dim contentType as string  
contentType = Response.ContentType
```

HttpResponse.Cookies

Syntax

```
HttpCookieCollection Cookies
```

Description

The Cookies property contains a collection of all cookies to be transmitted to the client in the current response.

Example

Listings 20.143 and 20.144 demonstrate how to send a new cookie to the client.

Listing 20.143 Using the Cookies Property (C#)

```
HttpCookie newCookie = new HttpCookie("FullName");
newCookie.Value = "Rachel Medina";
Response.Cookies.Add(newCookie);
```

Listing 20.144 Using the Cookies Property (Visual Basic.NET)

```
dim newCookie as HttpCookie

newCookie = new HttpCookie("FullName")
newCookie.Value = "Rachel Medina"
Response.Cookies.Add(newCookie)
```

HttpResponse.Expires

Syntax

```
Int32 Expires
```

Description

The Expires property contains the amount of time, in minutes, before the response expires.

Example

Listings 20.145 and 20.146 demonstrate how to specify the expiration time of a page as an hour.

Listing 20.145 Using the Expires Property (C#)

```
Response.Expires = 60;
```

Listing 20.146 Using the Expires Property (Visual Basic.NET)

```
Response.Expires = 60
```

HttpResponse.ExpiresAbsolute

Syntax

```
DateTime ExpiresAbsolute
```

Description

The **ExpiresAbsolute** property specifies the date and time at which a page will expire.

Example

Listings 20.147 and 20.148 demonstrate how to specify the absolute expiration date for a page.

Listing 20.147 Using the ExpiresAbsolute Property (C#)

```
Response.ExpiresAbsolute = DateTime.Now;
```

Listing 20.148 Using the ExpiresAbsolute Property (Visual Basic.NET)

```
Response.ExpiresAbsolute = DateTime.Now
```

HttpResponse.IsClientConnected

Syntax

```
Boolean IsClientConnected
```

Description

The **IsClientConnected** property contains True if the client is still connected to the Web server during the processing of a response.

Example

Listings 20.149 and 20.150 demonstrate how to see if a client is still connected during the processing of a response.

Listing 20.149 Using the IsClientConnected Property (C#)

```
if( Response.IsClientConnected ) {  
    //process the next long query since we know client is connected  
}
```

Listing 20.150 Using the IsClientConnected Property (Visual Basic.NET)

```
if Response.IsClientConnected then  
    'process the next long query since we know client is connected  
end if
```

HttpResponse.Status

Syntax

```
String Status
```

Description

The **Status** property contains the HTTP status that is sent to the client.

Example

Listings 20.151 and 20.152 demonstrate how to retrieve the HTTP status.

Listing 20.151 Using the status Property (C#)

```
string status = Response.Status;
```

Listing 20.152 Using the status Property (Visual Basic.NET)

```
dim status as string  
status = Response.Status
```

HttpResponse.StatusCode

Syntax

```
Int32 StatusCode
```

Description

The StatusCode property contains the HTTP status code sent to the client.

Example

Listings 20.153 and 20.154 demonstrate how to retrieve the HTTP status code sent to the client.

Listing 20.153 Using the StatusCode Property (C#)

```
int statusCode = Response.StatusCode;
```

Listing 20.154 Using the StatusCode Property (Visual Basic.NET)

```
dim statusCode as Int32  
statusCode = Response.StatusCode
```

HttpResponse.StatusDescription

Syntax

```
String StatusDescription
```

Description

The StatusDescription property contains the description of the HTTP status sent to the client.

Example

Listings 20.155 and 20.156 demonstrate how to retrieve the HTTP status description sent to the client.

Listing 20.155 Using the StatusDescription Property (C#)

```
string statusDescription = Response.StatusDescription;
```

Listing 20.156 Using the StatusDescription Property (Visual Basic.NET)

```
dim statusDescription as string  
statusDescription = Response.StatusDescription
```

HttpResponse.SuppressContent

Syntax

```
Boolean SuppressContent
```

Description

If the `SuppressContent` property contains `True`, HTTP output will not be sent to the client.

Example

Listings 20.157 and 20.158 demonstrate how to set the `SuppressContent` property.

Listing 20.157 Using the SuppressContent Property (C#)

```
Response.SuppressContent = true;
```

Listing 20.158 Using the SuppressContent Property (Visual Basic.NET)

```
Response.SuppressContent = true
```

HttpResponse.AddHeader()

Syntax

```
Void AddHeader( String name, String value )
```

Description

The `AddHeader()` method adds a new header to the current response.

Example

Listings 20.159 and 20.160 demonstrate how to add a new header to the output stream.

Listing 20.159 Using the AddHeader() Method (C#)

```
Response.AddHeader( "NewHeader", "NewHeaderValue" );
```

Listing 20.160 Using the AddHeader() Method (Visual Basic.NET)

```
Response.AddHeader( "NewHeader", "NewHeaderValue" )
```

HttpResponse.AppendCookie()

Syntax

```
Void AppendCookie( HttpCookie cookie )
```

Description

The AppendCookie() method adds a cookie to the cookie collection.

Example

Listings 20.161 and 20.162 demonstrate how to create a new cookie, set its value, and then add it to the cookie collection.

Listing 20.161 Using the AppendCookie() Method (C#)

```
HttpCookie newCookie = new HttpCookie( "UserName" );
newCookie.Value = "Banquo";
Response.AppendCookie( newCookie );
```

Listing 20.162 Using the AppendCookie() Method (Visual Basic.NET)

```
dim newCookie as new HttpCookie("UserName")
newCookie.Value = "Banquo"
Response.AppendCookie(newCookie)
```

HttpResponse.AppendHeader()

Syntax

```
Void AppendHeader()
```

Description

The AppendHeader() method adds a new header to the collection of headers sent to the client. An exception error occurs if this method is used after the headers have already been sent. Because the entire HTTP response is buffered until all server-side processing is complete, this exception occurs only if the default behavior is changed.

Example

Listings 20.163 and 20.164 demonstrate how to add a new header to the response.

Listing 20.163 Using the AppendHeader() Method (C#)

```
Response.AppendHeader("NewHeader", "NewHeaderValue");
```

Listing 20.164 Using the AppendHeader() Method (Visual Basic.NET)

```
Response.AppendHeader("NewHeader", "NewHeaderValue")
```

HttpResponse.AppendToLog()

Syntax

```
Void AppendToLog( String param )
```

Description

The AppendToLog() method adds a custom entry to the IIS log.

Example

Listings 20.165 and 20.166 demonstrate how to add a custom entry to the IIS log file.

Listing 20.165 Using the AppendToLog() Method (C#)

```
Response.AppendToLog("Custom Log Entry Information");
```

Listing 20.166 Using the AppendToLog() Method (Visual Basic.NET)

```
Response.AppendToLog("Custom Log Entry Information")
```

HttpResponse.BinaryWrite()

Syntax

```
Void BinaryWrite( Byte[] buffer )
```

Description

The BinaryWrite() method writes a string of binary characters to the HTTP output stream.

Example

Listings 20.167 and 20.168 demonstrate how to read the contents of a file and then output the stream directly to the client.

Listing 20.167 Using the BinaryWrite() Method (C#)

```
FileStream fs = new FileStream( "c:\\example.txt", FileMode.Open );
long FileSize = fs.Length;

byte[] Buffer = new byte[(int)FileSize];
fs.Read(Buffer, 0,(int)FileSize);
fs.Close();

Response.Write("Contents of File: ");
Response.BinaryWrite(Buffer);
```

Listing 20.168 Using the BinaryWrite() Method (Visual Basic.NET)

```
Dim fs As FileStream
dim FileSize As Long
```

Listing 20.168 continued

```
fs = New FileStream("c:\example.txt", FileMode.Open)
FileSize = fs.Length
Dim Buffer(CInt(FileSize)) As Byte
fs.Read(Buffer, 0, CInt(FileSize))
fs.Close()

Response.Write("Contents of File: ");
Response.BinaryWrite(Buffer);
```

HttpResponse.Clear()

Syntax

```
Void Clear()
```

Description

The `Clear()` method removes all content from the response buffer.

Example

Listings 20.169 and 20.170 demonstrate how to remove all content from the response buffer.

Listing 20.169 Using the `Clear()` Method (C#)

```
Response.Clear();
```

Listing 20.170 Using the `Clear()` Method (Visual Basic.NET)

```
Response.Clear()
```

HttpResponse.ClearContent()

Syntax

```
Void ClearContent()
```

Description

The `ClearContent()` method removes all content from the response buffer.

Example

Listings 20.171 and 20.172 demonstrate how to remove all content from the response buffer.

Listing 20.171 Using the `ClearContent()` Method (C#)

```
Response.ClearContent();
```

Listing 20.172 Using the `ClearContent()` Method (Visual Basic.NET)

```
Response.ClearContent()
```

HttpResponse.ClearHeaders()

Syntax

```
Void ClearHeaders()
```

Description

The `ClearHeaders()` method removes all headers from the response buffer.

Example

Listings 20.173 and 20.174 demonstrate how to remove all headers from the response buffer.

Listing 20.173 Using the `ClearHeaders()` Method (C#)

```
Response.ClearHeaders();
```

Listing 20.174 Using the `ClearHeaders()` Method (Visual Basic.NET)

```
Response.ClearHeaders()
```

HttpResponse.Close()

Syntax

```
Void Close()
```

Description

The `Close()` method closes the connection to the client.

Example

Listings 20.175 and 20.176 demonstrate how to close the connection to the client.

Listing 20.175 Using the `close()` Method (C#)

```
Response.Close();
```

Listing 20.176 Using the `close()` Method (Visual Basic.NET)

```
Response.Close
```

HttpResponse.End()

Syntax

```
Void End()
```

Description

The `End()` method flushes the response buffer and then closes the connection to the client.

Example

Listings 20.177 and 20.178 demonstrate how to end the current response.

Listing 20.177 Using the End() Method (C#)

```
Response.End();
```

Listing 20.178 Using the End() Method (Visual Basic.NET)

```
Response.End()
```

HttpResponse.Flush()

Syntax

```
Void Flush()
```

Description

The `Flush()` method sends all information that is currently in the response buffer to the client.

Example

Listings 20.179 and 20.180 demonstrate how to flush the response buffer.

Listing 20.179 Using the Flush() Method (C#)

```
Response.Flush();
```

Listing 20.180 Using the Flush() Method (Visual Basic.NET)

```
Response.Flush()
```

HttpResponse.Pics()

Syntax

```
Void Pics( String value )
```

Description

The `Pics()` method adds a PICS-Label header to the current response.

Example

Listings 20.181 and 20.182 demonstrate how to add a PICS-Label header to the current response. The PICS-Label header is a World Wide Web (W3) Consortium standard that will be the basis for a content-rating system in the future.

Listing 20.181 Using the Pics() Method (C#)

```
Response.Pics("All-Ages");
```

Listing 20.182 Using the Pics() Method (Visual Basic.NET)

```
Response.Pics("All-Ages")
```

HttpResponse.Redirect()

Syntax

```
Void Redirect( String url )
```

Description

The `Redirect()` method sends the client to a new URL. Because the `Redirect()` method actually performs the redirection through a `META` tag, if the headers have already been sent to the client, an exception error is generated.

Example

Listings 20.183 and 20.184 demonstrate how to automatically send the client to a new URL.

Listing 20.183 Using the Redirect() Method (C#)

```
Response.Redirect("http://www.bullseyeart.com");
```

Listing 20.184 Using the Redirect() Method (Visual Basic.NET)

```
Response.Redirect("http://www.bullseyeart.com")
```

HttpResponse.Write()

Syntax

```
Void Write( Char[] buffer, Int32 index, Int32 count )
Void Write( Char ch )
Void Write( Object obj )
Void Write( String s )
```

Description

The `Write()` method writes information to the output stream for the current request.

Example

Listings 20.185 and 20.186 demonstrate how to insert a string into the output stream.

Listing 20.185 Using the Write() Method (C#)

```
Response.Write("Is there anybody in there?");
```

Listing 20.186 Using the Write() Method (Visual Basic.NET)

```
Response.Write("Is there anybody in there?")
```

HttpResponse.WriteFile()

Syntax

```
Void WriteFile( IntPtr fileHandle, Int64 offset, Int64 size )
Void WriteFile( String filename )
Void WriteFile( String filename, Boolean readIntoMemory )
Void WriteFile( String filename, Int64 offset, Int64 size )
```

Description

The WriteFile() method writes the contents of a file directly to the output stream for a response.

Example

Listings 20.187 and 20.188 demonstrate how to output the contents of a file directly to the response.

Listing 20.187 Using the WriteFile() Method (C#)

```
Response.WriteFile("c:\\example.txt");
```

Listing 20.188 Using the WriteFile() Method (Visual Basic.NET)

```
Response.WriteFile("c:\\example.txt")
```

CHAPTER 21

System.Web.UI. WebControls Reference

All Web controls in ASP.NET are in the `System.Web.UI.WebControls` namespace. This chapter examines the properties of selected Web controls in depth. Specifically, the `WebControl` base class, various Web controls, and validation controls are covered.

The `WebControl` Class

All Web controls are derived from a common base class, `WebControl`, to ensure a consistent object model. The properties in this base class are covered in this section. These properties are present in any control derived from the `WebControl` class.

Table 21.1 Properties of the `WebControl` Class

Item	Description
<code>AccessKey</code>	Specifies the keyboard shortcut key used to access the Web control
<code>Attributes</code>	Contains a collection of attributes that do not have corresponding properties
<code>BackColor</code>	Specifies the background color
<code>BorderColor</code>	Specifies the border color
<code>BorderStyle</code>	Specifies the border style
<code>BorderWidth</code>	Specifies the border width
<code>ClientID</code>	Contains the ID of the Web control

Table 21.1 *continued*

Item	Description
CssClass	Specifies the cascading style sheets (CSS) class used to display the control
Enabled	Contains True if the Web control is enabled
EnableViewState	Specifies whether ViewState management should be on or off for the Web control
Font	Contains font information for the Web control
ForeColor	Contains the color of the foreground in the Web control
Height	Contains the height of the Web control
ID	Contains the ID of the Web control
Page	Contains a reference to the page object that is hosting the control
Parent	Contains a reference to the parent control
TabIndex	Contains the tab index for the Web control
ToolTip	Specifies the ToolTip text for the Web control
UniqueID	Contains the unique ID for the Web control
Visible	Contains True if the Web control is visible
Width	Specifies the width of a Web control

WebControl.AccessKey

Syntax

String AccessKey

Description

The `WebControl.AccessKey` property is used to specify the keyboard key used to access the Web control at runtime.

Example

Listings 21.1 and 21.2 demonstrate how to specify the access key for a Web control.

Listing 21.1 Using the AccessKey Property (C#)

```
MyWebControl.AccessKey = "d";
```

Listing 21.2 Using the AccessKey Property (Visual Basic.NET)

```
MyWebControl.AccessKey = "d"
```

WebControl.Attributes

Syntax

AttributeCollection Attributes

Description

The `WebControl.Attributes` property contains a collection of control attributes that do not have corresponding properties. Attributes can be used to pass custom attributes to the Web control.

Example

Listings 21.3 and 21.4 demonstrate how to add a new attribute to the attribute collection of a Web control. When the Web control renders, it will have "newAttribute="newValue" as an attribute in the rendered HTML.

Listing 21.3 Using the Attributes Property (C#)

```
MyWebControl.Attributes.Add("newAttribute", "newValue");
```

Listing 21.4 Using the Attributes Property (Visual Basic.NET)

```
MyWebControl.Attributes.Add("newAttribute", "newValue")
```

WebControl.BackColor

Syntax

```
Color BackColor
```

Description

The `WebControl.BackColor` property contains the background color of the Web control.

Example

Listings 21.5 and 21.6 demonstrate how to use the `BackColor` property to change the background color of a Web control.

Listing 21.5 Using the BackColor Property (C#)

```
MyWebControl.BackColor = System.Drawing.Color.Blue;
```

Listing 21.6 Using the BackColor Property (Visual Basic.NET)

```
MyWebControl.BackColor = System.Drawing.Color.Blue
```

WebControl.BorderColor

Syntax

```
Color BorderColor
```

Description

The `WebControl.BorderColor` property contains the color of the border of the Web control.

Example

Listings 21.7 and 21.8 demonstrate how to use the `BorderColor` property to change the border color of a Web control.

Listing 21.7 Using the `BorderColor` Property (C#)

```
MyWebControl.BorderColor = System.Drawing.Color.Blue;
```

Listing 21.8 Using the `BorderColor` Property (Visual Basic.NET)

```
MyWebControl.BorderColor = System.Drawing.Color.Blue
```

WebControl.BorderStyle

Syntax

```
BorderStyle BorderStyle
```

Description

The `WebControl.BorderStyle` property specifies a CSS style to apply to the borders of the Web control.

Example

Listings 21.9 and 21.10 apply a style named `question` to the borders of a Web control.

Listing 21.9 Using the `BorderStyle` Property (C#)

```
MyWebControl.BorderStyle = "question";
```

Listing 21.10 Using the `BorderStyle` Property (Visual Basic.NET)

```
MyWebControl.BorderStyle = "question"
```

WebControl.BorderWidth

Syntax

```
Unit BorderWidth
```

Description

The `WebControl.BorderWidth` property specifies the width of the border of the Web control.

Example

Listings 21.11 and 21.12 demonstrate how to specify a border width of 4 pixels for a Web control.

Listing 21.11 Using the BorderWidth Property (C#)

```
MyWebControl.BorderWidth = 4;
```

Listing 21.12 Using the BorderWidth Property (Visual Basic.NET)

```
MyWebControl.BorderWidth = 4
```

WebControl.ClientID

Syntax

```
String ClientID
```

Description

The `WebControl.ClientID` property contains the `ID` attribute of the Web control. This is the ID that the developer assigns to the control so that it can be referenced programmatically.

Example

Listings 21.13 and 21.142 demonstrate how to retrieve the ID of a Web control.

Listing 21.13 Using the clientID Property (C#)

```
msg.Text = MyWebControl.ClientID;
```

Listing 21.14 Using the clientID Property (Visual Basic.NET)

```
msg.Text = MyWebControl.ClientID
```

WebControl.CssClass

Syntax

```
String CssClass
```

Description

The `WebControl.CssClass` property specifies a CSS class to apply to the control.

Example

Listings 21.15 and 21.16 demonstrate how to use the `CssClass` property to assign a style sheet to the control.

Listing 21.15 Using the cssClass Property (C#)

```
MyWebControl.CssClass = "main";
```

Listing 21.16 Using the cssClass Property (Visual Basic.NET)

```
MyWebControl.CssClass = "main"
```

WebControl.Enabled

Syntax

Boolean Enabled

Description

The WebControl.Enabled property contains True if the control is currently enabled. To disable a control, you set the property to False.

Example

Listings 21.17 and 21.18 demonstrate how to disable a Web control by using the Enabled property.

Listing 21.17 Using the Enabled Property (C#)

```
MyWebControl.Enabled = false;
```

Listing 21.18 Using the Enabled Property (Visual Basic.NET)

```
MyWebControl.Enabled = false
```

WebControl.EnableViewState

Syntax

Boolean EnableViewState

Description

The WebControl.EnableViewState property contains True if ViewState management is enabled for the Web control. To disable ViewState management for the control, set this property to False.

Example

Listings 21.19 and 21.20 demonstrate how to disable ViewState management for a Web control.

Listing 21.19 Using the EnableViewState Property (C#)

```
MyWebControl.EnableViewState = false;
```

Listing 21.20 Using the EnableViewState Property (Visual Basic.NET)

```
MyWebControl.EnableViewState = false
```

WebControl.Font

Syntax

FontInfo Font

Description

The WebControl.Font property contains information about the font of the Web control.

Example

Listings 21.21 and 21.22 demonstrate how to use the Font property of a Web control to make the font of that Web control bold.

Listing 21.21 Using the Font Property (C#)

```
MyWebControl.Font.Bold = true;
```

Listing 21.22 Using the Font Property (Visual Basic.NET)

```
MyWebControl.Font.Bold = true
```

WebControl.ForeColor

Syntax

Color ForeColor

Description

The WebControl.ForeColor property contains the color of the foreground in the Web control.

Example

Listings 21.23 and 21.24 demonstrate how to use the ForeColor property of a Web control to make the foreground yellow.

Listing 21.23 Using the ForeColor Property (C#)

```
MyWebControl.ForeColor = System.Drawing.Color.Yellow;
```

Listing 21.24 Using the ForeColor Property (Visual Basic.NET)

```
MyWebControl.ForeColor = System.Drawing.Color.Yellow
```

WebControl.Height

Syntax

Unit Height

Description

The WebControl.Height property specifies the height of the Web control.

Example

Listings 21.25 and 21.26 demonstrate how to use the Height property to set the height of a Web control to 100 pixels.

Listing 21.25 Using the Height Property (C#)

```
MyWebControl.Height = 100;
```

Listing 21.26 Using the Height Property (Visual Basic.NET)

```
MyWebControl.Height = 100
```

WebControl.ID

Syntax

String ID

Description

The WebControl.ID property contains the ID attribute of a Web control.

Example

Listings 21.27 and 21.28 demonstrate how to retrieve the ID of a Web control.

Listing 21.27 Using the ID Property (C#)

```
msg.Text = MyWebControl.ID;
```

Listing 21.28 Using the ID Property (Visual Basic.NET)

```
dimsg.Text = MyWebControl.ID
```

WebControl.Page

Syntax

Page Page

Description

The WebControl.Page property contains a reference to the page object that is hosting the Web control.

Example

Listings 21.29 and 21.30 demonstrate how to get an instance of the current page that is hosting the Web control.

Listing 21.29 Using the Page Property (C#)

```
Page currentPage = MyWebControl.Page;
```

Listing 21.30 Using the Page Property (Visual Basic.NET)

```
Dim currentPage as new Page()  
currentPage = MyWebControl.Page
```

WebControl.Parent

Syntax

Control Parent

Description

The `WebControl.Parent` property contains a reference to the parent control of the Web control.

Example

Listings 21.31 and 21.32 demonstrate how to use the `Parent` property to get an instance of the parent control of the current Web control.

Listing 21.31 Using the Parent Property (C#)

```
Control newControl = MyWebControl.Parent;
```

Listing 21.32 Using the Parent Property (Visual Basic.NET)

```
Dim newControl = New Control()  
newControl = MyWebControl.Parent
```

WebControl.TabIndex

Syntax

Int16 TabIndex

Description

The `WebControl.TabIndex` property contains an integer value that is used to determine the Tab index between the controls on a page. (The *Tab index* controls the order of focus between fields on a Web form as the user presses the Tab key.)

Example

Listings 21.33 and 21.34 demonstrate how to change the Tab index of a Web control to 4.

Listing 21.33 Using the TabIndex Property (C#)

```
MyWebControl.TabIndex = 4;
```

Listing 21.34 Using the TabIndex Property (Visual Basic.NET)

```
MyWebControl.TabIndex = 4
```

WebControl.ToolTip

Syntax

String ToolTip

Description

The `WebControl.ToolTip` property specifies a string that will be displayed when the user places the mouse pointer over the rendered control. This text only appears in browsers that support this feature.

Example

Listings 21.35 and 21.36 demonstrate how to set the `ToolTip` of a Web control.

Listing 21.35 Using the ToolTip Property (C#)

```
MyWebControl.ToolTip = "Please enter text into this field";
```

Listing 21.36 Using the ToolTip Property (Visual Basic.NET)

```
MyWebControl.ToolTip = "Please enter text into this field"
```

WebControl.Visible

Syntax

Boolean Visible

Description

The `WebControl.Visible` property specifies whether a control is visible to the user.

Example

Listings 21.37 and 21.38 demonstrate how to use the `Visible` property to hide a Web control from the user.

Listing 21.37 Using the Visible Property (C#)

```
MyWebControl.Visible = false;
```

Listing 21.38 Using the Visible Property (Visual Basic.NET)

```
MyWebControl.Visible = false
```

WebControl.Width

Syntax

Unit Width

Description

The WebControl.Width property specifies the width of a Web control.

Example

Listings 21.39 and 21.40 demonstrate how to use the Width property to set the width of a Web control to 400 pixels.

Listing 21.39 Using the Width Property (C#)

```
MyWebControl.Width = 400;
```

Listing 21.40 Using the Width Property (Visual Basic.NET)

```
MyWebControl.Width = 400
```

The HyperLink Class

The HyperLink Web control is used to place a hyperlink on the rendered page. In addition to the properties in the WebControl base class, the HyperLink Web control also has the properties covered in this section.

Table 21.2 Properties of the HyperLink Class

Item	Description
ImageUrl	Specifies the URL of an image to display as the rendered hyperlink
NavigateUrl	Specifies the URL that will be loaded when the user clicks on the hyperlink
Target	Specifies the target window or frame in which the target of the hyperlink will be loaded

HyperLink.ImageUrl

Syntax

```
String ImageUrl
```

Description

The HyperLink.ImageUrl property is used to specify the URL of an image that will be displayed as the rendered hyperlink.

Example

Listings 21.41 and 21.42 demonstrate how to specify the URL of an image for a hyperlink.

Listing 21.41 Using the ImageUrl Property (C#)

```
MyHyperLink.ImageUrl = "/images/test.jpg";
```

Listing 21.42 Using the ImageUrl Property (Visual Basic.NET)

```
MyHyperLink.ImageUrl = "/images/test.jpg"
```

HyperLink.NavigateUrl

Syntax

```
String NavigateUrl
```

Description

The HyperLink.NavigateUrl property is used to specify the URL that will be loaded when the user clicks on the rendered hyperlink.

Example

Listings 21.43 and 21.44 demonstrate how to specify the URL that will be loaded when the user clicks on the rendered hyperlink.

Listing 21.43 Using the NavigateUrl Property (C#)

```
MyHyperLink.ImageUrl = "http://www.theonion.com";
```

Listing 21.44 Using the NavigateUrl Property (Visual Basic.NET)

```
MyHyperLink.ImageUrl = "http://www.theonion.com"
```

HyperLink.Target

Syntax

```
String Target
```

Description

The HyperLink.Target property specifies the target window or frame in which the target of the hyperlink will be loaded.

Example

Listings 21.45 and 21.46 demonstrate how to specify a new browser window for the hyperlink target.

Listing 21.45 Using the Target Property (C#)

```
MyHyperLink.Target = "_blank";
```

Listing 21.46 Using the Target Property (Visual Basic.NET)

```
MyHyperLink.Target = "_blank"
```

HyperLink.Text

Syntax

`String Text`

Description

The HyperLink.Text property contains the text to display for a hyperlink.

Example

Listings 21.47 and 21.48 demonstrate how to change the text of a hyperlink programmatically.

Listing 21.47 Using the Text Property (C#)

```
MyHyperLink.Text = "Click here";
```

Listing 21.48 Using the Text Property (Visual Basic.NET)

```
MyHyperLink.Text = "Click here"
```

The Button Class

The Button Web control is used to place a standard HTML button on the rendered page. In addition to the properties in the WebControl base class, the Button Web control also has the properties covered in this section.

Table 21.3 Properties of the Button Class

Item	Description
CommandArgument	Sends additional arguments to the event processor
CommandName	Sends the name of the command to be performed to the event processor
Text	Specifies text that will appear on the rendered button

Button.CommandArgument

Syntax

`String CommandArgument`

Description

The Button.CommandArgument property is used to pass additional information to the event that processes the click of the button.

Example

Listings 21.49 and 21.50 demonstrate how to use the `CommandArgument` property to send additional arguments to the method that is processing the click event. The `CommandArgument` property is normally used in conjunction with the `CommandName` property. For this reason, the examples for each property are the same.

Listing 21.49 Using the CommandArgument Property (C#)

```
MyButton.CommandName = "Sort";
MyButton.CommandArgument = "ByName";
```

Listing 21.50 Using the CommandArgument Property (Visual Basic.NET)

```
MyButton.CommandName = "Sort"
MyButton.CommandArgument = "ByName"
```

Button.CommandName

Syntax

String CommandName

Description

The `Button.CommandName` property passes the name of the command to be performed to the button's click event.

Example

Listings 21.51 and 21.52 demonstrate how to use the `CommandName` property in conjunction with the `CommandArgument` property in order to send additional information to the method that is processing the click event of the button.

Listing 21.51 Using the CommandName Property (C#)

```
MyButton.CommandName = "Sort";
MyButton.CommandArgument = "ByName";
```

Listing 21.52 Using the CommandName Property (Visual Basic.NET)

```
MyButton.CommandName = "Sort"
MyButton.CommandArgument = "ByName";
```

Button.Text

Syntax

String Text

Description

The `Button.Text` property specifies text that will appear on the rendered button.

Example

Listings 21.53 and 21.54 demonstrate how to change the text on a button programmatically.

Listing 21.53 Using the Text Property (C#)

```
MyButton.Text = "Click Here To Submit Form";
```

Listing 21.54 Using the Text Property (Visual Basic.NET)

```
MyButton.Text = "Click Here To Submit Form"
```

The Calendar Class

The Calendar Web control is larger and more powerful than the other Web controls covered in this chapter. By placing a simple tag in a Web form, you can render an HTML-based calendar that can accept user input.

The Calendar Web control is instantiated like this:

```
<asp:Calendar id="MyCalendar" runat="server"></asp:Calendar>
```

This section covers the properties of the Calendar Web control.

Table 21.4 Properties of the Calendar Class

Item	Description
CellPadding	Specifies the padding distance inside cells in the Calendar control
CellSpacing	Specifies the amount of spacing between cells in the Calendar control
DayHeaderStyle	Specifies style and format information for the day headers
DayNameFormat	Specifies the format of the day names
DayStyle	Specifies style and format information for the days in the month
FirstDayOfWeek	Specifies the first calendar day of the week
NextMonthText	Specifies the text to display in the hyperlink to show next month
NextPrevFormat	Specifies the format of the next and previous month navigation elements
NextPrevStyle	Specifies style and format information to apply to the next and previous month navigation elements
OtherMonthDayStyle	Specifies style and format information for days on the calendar that are not in the current month
PrevMonthText	Specifies the text of the hyperlink which shows the previous month
SelectedDate	Specifies the currently selected date

Table 21.4 continued

Item	Description
SelectedDayStyle	Specifies the style and format of the day the user has selected
SelectionMode	Specifies the selection choices a user has available
SelectMonthText	Specifies the text of the hyperlink that enables a user to select an entire month
SelectorStyle	Contains the style and format information for the week and month selection elements
SelectWeekText	Specifies the text to be displayed next to the weeks that will enable a user to select a week at a time
ShowDayHeader	Specifies whether the header row on the Calendar control should be displayed
ShowGridLines	Specifies whether the Calendar grid lines should be rendered
ShowNextPrevMonth	Specifies whether the next and previous month links should appear
ShowTitle	Specifies whether the Calendar title is rendered
TitleFormat	Specifies the format of the title
TitleStyle	Contains style and format information for the Calendar title
TodayDayStyle	Contains style and format information for the current day
TodaysDate	Specifies the current date
VisibleDate	Specifies the date used to determine which month to display on the rendered calendar

Calendar.CellPadding

Syntax

Int32 CellPadding

Description

The `Calendar.CellPadding` property changes the padding distance between cells in the `Calendar` control.

Example

Listings 21.55 and 21.56 demonstrate how to use the `CellPadding` property to specify a padding of 4 to each cell in the `Calendar` control.

Listing 21.55 Using the CellPadding Property (C#)

```
MyCalendar.CellPadding = 4;
```

Listing 21.56 Using the CellPadding Property (Visual Basic.NET)

```
MyCalendar.CellPadding = 4
```

Calendar.CellSpacing

Syntax

```
Int32 CellSpacing
```

Description

The `Calendar.CellSpacing` property specifies the amount of spacing between cells in the `Calendar` control.

Example

Listings 21.57 and 21.58 demonstrate how to use the `CellSpacing` property to set the spacing between cells to 4 pixels.

Listing 21.57 Using the CellSpacing Property (C#)

```
MyCalendar.CellSpacing = 4;
```

Listing 21.58 Using the CellSpacing Property (Visual Basic.NET)

```
MyCalendar.CellSpacing = 4;
```

Calendar.DayHeaderStyle

Syntax

```
TableItemStyle DayHeaderStyle
```

Description

The `Calendar.DayHeaderStyle` property specifies style and format information for the day headers at the top of the `Calendar` control.

Example

Listings 21.59 and 21.60 demonstrate how to use the `DayHeaderStyle` property to specify a CSS class for the row.

Listing 21.59 Using the DayHeaderStyle Property (C#)

```
MyCalendar.DayHeaderStyle.CssClass = "main";
```

Listing 21.60 Using the DayHeaderStyle Property (Visual Basic.NET)

```
MyCalendar.DayHeaderStyle.CssClass = "main"
```

Calendar.DayNameFormat

Syntax

```
DayNameFormat DayNameFormat
```

Description

The `Calendar.DayNameFormat` property specifies the format of the day names as rendered by the `Calendar` control. Valid values are `FirstLetter`, `FirstTwoLetters`, `Full`, and `Short`. `Short` is the default value.

Example

Listings 21.61 and 21.62 demonstrate how to use the `DayNameFormat` property to display day names, using their full name.

Listing 21.61 Using the DayNameFormat Property (C#)

```
MyCalendar.DayNameFormat = DayNameFormat.Full;
```

Listing 21.62 Using the DayNameFormat Property (Visual Basic.NET)

```
dim browCap as new Boolean  
browCap = Request.Browser.ActiveXControls
```

Calendar.DayStyle

Syntax

```
TableItemStyle DayStyle
```

Description

The `Calendar.DayStyle` property specifies style and format information for the days in the month.

Example

Listings 21.63 and 21.64 demonstrate how to specify a CSS class for the days in the `Calendar` control.

Listing 21.63 Using the DayStyle Property (C#)

```
MyCalendar.DayStyle.CssClass = "main";
```

Listing 21.64 Using the DayStyle Property (Visual Basic.NET)

```
MyCalendar.DayStyle.CssClass = "main"
```

Calendar.FirstDayOfWeek

Syntax

```
FirstDayOfWeek FirstDayOfWeek
```

Description

The `Calendar.FirstDayOfWeek` property specifies the first calendar day of the week. Sunday is the default first day of the week.

Example

Listings 21.65 and 21.66 demonstrate how to specify Wednesday as the first day of the calendar week.

Listing 21.65 Using the FirstDayOfWeek Property (C#)

```
MyCalendar.FirstDayOfWeek = FirstDayOfWeek.Wednesday;
```

Listing 21.66 Using the FirstDayOfWeek Property (Visual Basic.NET)

```
MyCalendar.FirstDayOfWeek = FirstDayOfWeek.Wednesday
```

Calendar.NextMonthText

Syntax

```
String NextMonthText
```

Description

The `Calendar.NextMonthText` property specifies the text to display in the header of the `Calendar` control that directs the user to click to the next month. By default, this property is blank.

Example

Listings 21.67 and 21.68 specify the text to display so that the user can click ahead to the next month.

Listing 21.67 Using the NextMonthText Property (C#)

```
MyCalendar.NextMonthText = "Go to next";
```

Listing 21.68 Using the NextMonthText Property (Visual Basic.NET)

```
MyCalendar.NextMonthText = "Go to next"
```

Calendar.NextPrevFormat

Syntax

NextPrevFormat NextPrevFormat

Description

The `Calendar.NextPrevFormat` property specifies the format of the next and previous month navigation elements. Valid values are `CustomText`, `FullMonth`, and `ShortMonth`. `FullMonth` displays the full month name and `ShortMonth` displays an abbreviation.

Example

Listings 21.69 and 21.70 demonstrate how to use the `NextPrevFormat` property to specify `FullMonth` for the format of the navigation elements.

Listing 21.69 Using the NextPrevFormat Property (C#)

```
MyCalendar.NextPrevFormat = FullMonth;
```

Listing 21.70 Using the NextPrevFormat Property (Visual Basic.NET)

```
MyCalendar.NextPrevFormat = FullMonth
```

Calendar.NextPrevStyle

Syntax

TableItemStyle NextPrevStyle

Description

The `Calendar.NextPrevStyle` property specifies style and format information to apply to the next and previous month navigation elements.

Example

Listings 21.71 and 21.72 demonstrate how to specify a CSS class header to the next and previous navigation elements.

Listing 21.71 Using the NextPrevStyle Property (C#)

```
MyCalendar.NextPrevStyle.CssClass = "header";
```

Listing 21.72 Using the NextPrevStyle Property (Visual Basic.NET)

```
MyCalendar.NextPrevStyle.CssClass = "header"
```

Calendar.OtherMonthDayStyle

Syntax

TableItemStyle OtherMonthDayStyle

Description

The `Calendar.OtherMonthDayStyle` property specifies style and format information for days on the calendar that are not in the current month.

Example

Listings 21.73 and 21.74 demonstrate how to specify a CSS class `OtherDay` to the next and previous navigation elements.

Listing 21.73 Using the `OtherMonthDayStyle` Property (C#)

```
MyCalendar.OtherMonthDayStyle.CssClass = "OtherDay";
```

Listing 21.74 Using the `OtherMonthDayStyle` Property (Visual Basic.NET)

```
MyCalendar.OtherMonthDayStyle.CssClass = "OtherDay"
```

Calendar.PrevMonthText

Syntax

```
String PrevMonthText
```

Description

The `Calendar.PrevMonthText` property specifies the text to display in the header of the `Calendar` control that directs the user to click to the previous month. By default, this property is blank.

Example

Listings 21.75 and 21.76 specify the text to display that will enable the user to click to the previous month.

Listing 21.75 Using the `PrevMonthText` Property (C#)

```
MyCalendar.NextMonthText = "Go to previous";
```

Listing 21.76 Using the `PrevMonthText` Property (Visual Basic.NET)

```
MyCalendar.NextMonthText = "Go to previous"
```

Calendar.SelectedDate

Syntax

```
DateTime SelectedDate
```

Description

The `Calendar.SelectedDate` property contains the currently selected date.

Example

Listings 21.77 and 21.78 demonstrate how to retrieve the date that the user has selected on the Calendar control.

Listing 21.77 Using the SelectedDate Property (C#)

```
DateTime MyDate = MyCalendar.SelectedDate;
```

Listing 21.78 Using the SelectedDate Property (Visual Basic.NET)

```
Dim MyDate as DateTime  
MyDate = MyCalendar.SelectedDate
```

Calendar.SelectedDayStyle

Syntax

```
TableItemStyle SelectedDayStyle
```

Description

The `Calendar.SelectedDayStyle` property specifies the style and format of the day the user has selected.

Example

Listings 21.79 and 21.80 demonstrate how to specify a CSS class `SelectedDay` for the selected date.

Listing 21.79 Using the SelectedDayStyle Property (C#)

```
MyCalendar.SelectedDayStyle.CssClass = "SelectedDay";
```

Listing 21.80 Using the SelectedDayStyle Property (Visual Basic.NET)

```
MyCalendar.SelectedDayStyle.CssClass = "SelectedDay"
```

Calendar.SelectionMode

Syntax

```
CalendarSelectionMode SelectionMode
```

Description

The `Calendar.SelectionMode` property specifies the selection choices a user has available. The user can select the day, week, or month, based on the `SelectionMode` property. Valid values are `Day`, `DayWeek`, `DayWeekMonth`, and `None`.

Example

Listings 21.81 and 21.82 demonstrate how to use `SelectionMode` to allow users to select only the day.

Listing 21.81 Using the SelectionMode Property (C#)

```
MyCalendar.SelectionMode = CalendarSelectionMode.Day;
```

Listing 21.82 Using the SelectionMode Property (Visual Basic.NET)

```
MyCalendar.SelectionMode = CalendarSelectionMode.DayWeek
```

Calendar.SelectMonthText

Syntax

```
String SelectMonthText
```

Description

The `Calendar.SelectMonthText` property enables you to specify text that will appear in the upper-left corner of the rendered calendar. When a user clicks on this text, the entire month will be selected. In order for this property to function, the `Calendar.SelectionMode` must be set to `CalendarSelectionMode.DayWeekMonth`.

Example

Listings 21.83 and 21.84 demonstrate how to use the `SelectMonthText` property to display a hyperlink that enables the user to select the entire month.

Listing 21.83 Using the SelectMonthText Property (C#)

```
MyCalendar.SelectionMode = CalendarSelectionMode.DayWeekMonth;  
MyCalendar.SelectMonthText = "select month";
```

Listing 21.84 Using the SelectMonthText Property (Visual Basic.NET)

```
MyCalendar.SelectionMode = CalendarSelectionMode.DayWeekMonth  
MyCalendar.SelectMonthText = "select month"
```

Calendar.SelectorStyle

Syntax

```
TableItemStyle SelectorStyle
```

Description

The `Calendar.SelectorStyle` property contains the style and format information for the week and month selection elements.

Example

Listings 21.85 and 21.86 demonstrate how to retrieve the CSS class that is applied to the week and month selection elements.

Listing 21.85 Using the SelectorStyle Property (C#)

```
string cssClass = MyCalendar.SelectorStyle.CssClass;
```

Listing 21.86 Using the SelectorStyle Property (Visual Basic.NET)

```
dim cssClass as string  
cssClass = MyCalendar.SelectorStyle.CssClass
```

Calendar.SelectWeekText

Syntax

String SelectWeekText

Description

The `Calendar.SelectWeekText` property specifies the text to be displayed next to the weeks that will enable a user to select those weeks.

Example

Listings 21.87 and 21.88 demonstrate how to specify the text that will appear next to the weeks to enable a user to select a week at a time.

Listing 21.87 Using the SelectWeekText Property (C#)

```
MyCalendar.SelectWeekText = ">>";
```

Listing 21.88 Using the SelectWeekText Property (Visual Basic.NET)

```
MyCalendar.SelectWeekText = ">>"
```

Calendar.ShowDayHeader

Syntax

Boolean ShowDayHeader

Description

The `Calendar.ShowDayHeader` property specifies whether the header row on the `Calendar` control should be displayed. The default is `True`.

Example

Listings 21.89 and 21.90 demonstrate how to hide the header row of a calendar control.

Listing 21.89 Using the ShowDayHeader Property (C#)

```
MyCalendar.ShowDayHeader = false;
```

Listing 21.90 Using the ShowDayHeader Property (Visual Basic.NET)

```
MyCalendar.ShowDayHeader = false
```

Calendar>ShowGridLines

Syntax

```
Boolean ShowGridLines
```

Description

The `Calendar>ShowGridLines` property specifies whether the calendar grid lines should be rendered. The default is True.

Example

Listings 21.91 and 21.92 demonstrate how to turn off the gridlines on the rendered calendar.

Listing 21.91 Using the ShowGridLines Property (C#)

```
MyCalendar>ShowGridLines = false;
```

Listing 21.92 Using the ShowGridLines Property (Visual Basic.NET)

```
MyCalendar>ShowGridLines = false
```

Calendar>ShowNextPrevMonth

Syntax

```
Boolean ShowNextPrevMonth
```

Description

The `Calendar>ShowNextPrevMonth` property specifies whether the next and previous month links should appear. The default is True.

Example

Listings 21.93 and 21.94 demonstrate how to turn off the next and previous month links on the rendered calendar.

Listing 21.93 Using the ShowNextPrevMonth Property (C#)

```
MyCalendar>ShowNextPrevMonth = false;
```

Listing 21.94 Using the ShowNextPrevMonth Property (Visual Basic.NET)

```
MyCalendar>ShowNextPrevMonth = false
```

Calendar.ShowTitle

Syntax

Boolean ShowTitle

Description

The `Calendar.ShowTitle` property specifies whether the calendar title is rendered. The default is `True`.

Example

Listings 21.95 and 21.96 demonstrate how to remove the title from the rendered calendar.

Listing 21.95 Using the ShowTitle Property (C#)

```
MyCalendar.ShowTitle = false;
```

Listing 21.96 Using the ShowTitle Property (Visual Basic.NET)

```
MyCalendar.ShowTitle = false
```

Calendar.TitleFormat

Syntax

TitleFormat TitleFormat

Description

The `Calendar.TitleFormat` property specifies the format of the title. Valid values are `Month` and `MonthYear`. The default is `MonthYear`.

Example

Listings 21.97 and 21.98 demonstrate how to change the format of the title to display the month only.

Listing 21.97 Using the TitleFormat Property (C#)

```
MyCalendar.TitleFormat = TitleFormat.Month;
```

Listing 21.98 Using the TitleFormat Property (Visual Basic.NET)

```
MyCalendar.TitleFormat = TitleFormat.Month
```

Calendar.TitleStyle

Syntax

TableItemStyle TitleStyle

Description

The `Calendar.TitleStyle` property contains the style and format information for the calendar's title.

Example

Listings 21.99 and 21.100 demonstrate how to retrieve the CSS class that is applied to the calendar title.

Listing 21.99 Using the TitleStyle Property (C#)

```
string MyClass = MyCalendar.TitleStyle.CssClass;
```

Listing 21.100 Using the TitleStyle Property (Visual Basic.NET)

```
dim MyClass as string  
MyClass = MyCalendar.TitleStyle.CssClass
```

`Calendar.TodayDayStyle`

Syntax

`TableItemStyle TodayDayStyle`

Description

The `Calendar.TodayDayStyle` property contains the style and format information for the current day in the calendar.

Example

Listings 21.101 and 21.102 demonstrate how to retrieve the CSS class applied to the current calendar day.

Listing 21.101 Using the TodayDayStyle Property (C#)

```
string MyClass = MyCalendar.TodayDayStyle.CssClass;
```

Listing 21.102 Using the TodayDayStyle Property (Visual Basic.NET)

```
dim MyClass as string  
MyClass = MyCalendar.TodayDayStyle.CssClass
```

`Calendar.TodaysDate`

Syntax

`DateTime TodaysDate`

Description

The `Calendar.TodaysDate` property specifies the current date.

Example

Listings 21.103 and 21.104 demonstrate how to retrieve the current date, using the `TodaysDate` property.

Listing 21.103 Using the TodaysDate Property (C#)

```
DateTime MyDate = MyCalendar.TodaysDate;
```

Listing 21.104 Using the TodaysDate Property (Visual Basic.NET)

```
dim MyDate as DateTime  
MyDate = MyCalendar.TodaysDate
```

Calendar.VisibleDate

Syntax

```
DateTime VisibleDate
```

Description

The `Calendar.VisibleDate` property specifies the date used to determine which month to display in the rendered calendar.

Example

Listings 21.105 and 21.106 demonstrate how to specify the visible date.

Listing 21.105 Using the VisibleDate Property (C#)

```
MyCalendar.VisibleDate = new DateTime(1952, 12, 7);
```

Listing 21.106 Using the VisibleDate Property (Visual Basic.NET)

```
dim MyDate as new DateTime(1952, 12, 7)  
MyCalendar.VisibleDate = MyDate
```

Calendar.WeekendDayStyle

Syntax

```
TableItemStyle WeekendDayStyle
```

Description

The `Calendar.WeekendDayStyle` property contains the style and format information of the weekend days in the rendered calendar.

Example

Listings 21.107 and 21.108 demonstrate how to retrieve the CSS class of the weekend days of the calendar.

Listing 21.107 Using the WeekendDayStyle Property (C#)

```
string MyClass = MyCalendar.WeekendDayStyle.CssClass;
```

Listing 21.108 Using the WeekendDayStyle Property (Visual Basic.NET)

```
dim MyClass as string
MyClass = MyCalendar.WeekendDayStyle.CssClass
```

The Label Class

The Label Web control is one of the simplest Web controls. It is used to display a message on the page. It has only one property that isn't present in the WebControl base class.

Table 21.5 Properties of the Label Class

Item	Description
Text	Specifies the text to be displayed when the control is rendered

Label.Text

Syntax

String Text

Description

The Label.Text property contains the text that will display when the control is rendered.

Example

Listings 21.109 and 21.110 demonstrate how to set the text of a label.

Listing 21.109 Using the Label.Text Property (C#)

```
MyLabel.Text = "This is some text";
```

Listing 21.110 Using the Label.Text Property (Visual Basic.NET)

```
MyLabel.Text = "This is some text"
```

The Image Class

The Image Web control is used to display an image on the page. The properties of the Image class that are not part of the WebControl base class are covered in this section.

Table 21.6 Properties of the Image Class

Item	Description
AlternateText	Specifies the text to be displayed if the user has images turned off
ImageAlign	Aligns the rendered image on the page
ImageUrl	Specifies the URL to display when the user clicks on the image

Image.AltAlternateText

Syntax

String AlternateText

Description

The `Image.AltAlternateText` property contains the text to be displayed if the user has images turned off or if the user's browser does not support images.

Example

Listings 21.111 and 21.112 demonstrate how to specify the alternate text of a rendered image.

Listing 21.111 Using the Image.AltAlternateText Property (C#)

```
MyImage.AltAlternateText = "My two cats";
```

Listing 21.112 Using the Image.AltAlternateText Property (Visual Basic.NET)

```
MyImage.AltAlternateText = "My two cats"
```

Image.ImageAlign

Syntax

ImageAlign ImageAlign

Description

The `Image.ImageAlign` property is used to align the rendered image on the page.

Example

Listings 21.113 and 21.114 demonstrate how to align an image to the left.

Listing 21.113 Using the Image.ImageAlign Property (C#)

```
MyImage.ImageAlign = ImageAlign.Left;
```

Listing 21.114 Using the Image.ImageAlign Property (Visual Basic.NET)

```
MyImage.ImageAlign = ImageAlign.Left
```

Image.ImageUrl**Syntax**

String ImageUrl

Description

The `Image.ImageUrl` property is used to specify a URL to display to the user when he or she clicks on the image.

Example

Listings 21.115 and 21.116 demonstrate how to specify the URL to load when the user clicks the image.

Listing 21.115 Using the Image.ImageUrl Property (C#)

```
MyImage.ImageUrl = "http://www.slashdot.org";
```

Listing 21.116 Using the Image.ImageUrl Property (Visual Basic.NET)

```
MyImage.ImageUrl = "http://www.slashdot.org"
```

The Panel Class

The `Panel` Web control renders a logical division to the browser enabling you to group elements of your interface together. The properties of the `Panel` class that are not in the `WebControl` base class are covered in this section.

Table 21.7 Properties of the Panel Class

Item	Description
<code>BackImageUrl</code>	Specifies an image to display in the background of the rendered panel
<code>HorizontalAlign</code>	Aligns the rendered panel horizontally within its container
<code>Wrap</code>	Specifies whether the content in the rendered panel should be wrapped

Panel.BackImageUrl**Syntax**

String BackImageUrl

Description

The `Panel.BackImageUrl` property specifies an image to display in the background of the rendered panel.

Example

Listings 21.117 and 21.118 demonstrate how to set the background image of a panel.

Listing 21.117 Using the Panel.BackImageUrl Property (C#)

```
MyPanel.BackImageUrl = "/images/sandra.jpg";
```

Listing 21.118 Using the Panel.BackImageUrl Property (Visual Basic.NET)

```
MyPanel.BackImageUrl = "/images/sandra.jpg"
```

Panel.HorizontalAlign

Syntax

```
HorizontalAlign HorizontalAlign
```

Description

The `Panel.HorizontalAlign` property is used to align the rendered panel horizontally in its container. Valid values are `Center`, `Justify`, `Left`, `NotSet` and `Right`.

Example

Listings 21.119 and 21.120 demonstrate how to align a rendered panel to the left.

Listing 21.119 Using the Panel.HorizontalAlign Property (C#)

```
MyPanel.HorizontalAlign = HorizontalAlign.Left;
```

Listing 21.120 Using the Panel.HorizontalAlign Property (Visual Basic.NET)

```
MyPanel.HorizontalAlign = HorizontalAlign.Left
```

Panel.Wrap

Syntax

```
Boolean Wrap
```

Description

The `Panel.Wrap` property specifies whether the content in the rendered panel should be wrapped.

Example

Listings 21.121 and 21.122 demonstrate how to make the text in a rendered panel wrap from line to line.

Listing 21.121 Using the Panel.Wrap Property (C#)

```
MyPanel.Wrap = true;
```

Listing 21.122 Using the Panel.Wrap Property (Visual Basic.NET)

```
MyPanel.Wrap = true
```

The TextBox Class

The `TextBox` Web control is used to gather input from the user. It has three modes: Multiline, password, and single line. These modes, in addition to its other properties, are covered in this section.

Table 21.8 Properties of the TextBox Class

Item	Description
<code>AutoPostBack</code>	Causes the page to automatically submit the data to the server when the contents of the textbox change
<code>Columns</code>	Specifies the width of the rendered text box, in characters
<code>MaxLength</code>	Specifies the maximum amount of characters the user can type into the rendered text box
<code>ReadOnly</code>	Makes the rendered text box read-only
<code>Rows</code>	Specifies the height of the rendered text box in characters
<code>Text</code>	Specifies any default text to appear in the rendered text box
<code>TextMode</code>	Specifies whether the type of text box rendered
<code>Wrap</code>	Specifies whether the content in the rendered text box should be wrapped

TextBox.AutoPostBack

Syntax

```
Boolean AutoPostBack
```

Description

If set to True, the `TextBox.AutoPostBack` property causes the page to automatically submit the data to the server when the contents change.

Example

Listings 21.123 and 21.124 demonstrate how to enable automatic postback of form data when the data in the text box changes.

Listing 21.123 Using the TextBox.AutoPostBack Property (C#)

```
MyTextBox.AutoPostBack = true;
```

Listing 21.124 Using the TextBox.AutoPostBack Property (Visual Basic.NET)

```
MyTextBox.AutoPostBack = true
```

TextBox.Columns

Syntax

Int32 Columns

Description

The `TextBox.Columns` property specifies the width of the rendered text box, in characters.

Example

Listings 21.125 and 21.126 demonstrate how to set the width of a text box to 80 characters.

Listing 21.125 Using the TextBox.Columns Property (C#)

```
MyTextBox.Columns = 80;
```

Listing 21.126 Using the TextBox.Columns Property (Visual Basic.NET)

```
MyTextBox.Columns = 80
```

TextBox.MaxLength

Syntax

Int32 MaxLength

Description

The `TextBox.MaxLength` property specifies the maximum amount of characters the user can type into the rendered text box.

Example

Listings 21.127 and 21.128 demonstrate how to set the maximum input length of a text box to 20 characters.

Listing 21.127 Using the TextBox.MaxLength Property (C#)

```
MyTextBox.MaxLength = 20;
```

Listing 21.128 Using the TextBox.MaxLength Property (Visual Basic.NET)

```
MyTextBox.MaxLength = 20
```

TextBox.ReadOnly

Syntax

```
Boolean ReadOnly
```

Description

The `TextBox.ReadOnly` property makes the rendered text box read-only if it is set to `True`.

Example

Listings 21.129 and 21.130 demonstrate how to make a text box read-only by using the `ReadOnly` property.

Listing 21.129 Using the TextBox.ReadOnly Property (C#)

```
MyTextBox.ReadOnly = true;
```

Listing 21.130 Using the TextBox.ReadOnly Property (Visual Basic.NET)

```
MyTextBox.ReadOnly = true
```

TextBox.Rows

Syntax

```
Int32 Rows
```

Description

The `TextBox.Rows` property specifies the height of the rendered text box, in characters.

Example

Listings 21.131 and 21.132 demonstrate how to set the height of a text box to 20 characters.

Listing 21.131 Using the TextBox.Rows Property (C#)

```
MyTextBox.Rows = 20;
```

Listing 21.132 Using the TextBox.Rows Property (Visual Basic.NET)

```
MyTextBox.Rows = 20
```

TextBox.Text

Syntax

String Text

Description

The TextBox.Text property specifies any default text to appear in the rendered text box.

Example

Listings 21.133 and 21.134 demonstrate how to set some default text to appear in the text box when the page is loaded.

Listing 21.133 Using the TextBox.Text Property (C#)

```
MyTextBox.Text = "Please enter your name here";
```

Listing 21.134 Using the TextBox.Text Property (Visual Basic.NET)

```
MyTextBox.Text = "Please enter your name here"
```

TextBox.TextMode

Syntax

TextBoxMode TextMode

Description

The TextBox.TextMode property specifies whether the rendered text box should be multiline, single line, or a password field.

Example

Listings 21.135 and 21.136 specify a text box as a password input text box. The user's input will be echoed to the screen as * characters.

Listing 21.135 Using the TextBox.TextMode Property (C#)

```
MyTextBox.TextMode = TextBoxMode.Password;
```

Listing 21.136 Using the TextBox.TextMode Property (Visual Basic.NET)

```
MyTextBox.TextMode = TextBoxMode.Password
```

TextBox.Wrap

Syntax

Boolean Wrap

Description

The `TextBox.Wrap` property specifies whether the contents of the rendered text box should be wrapped.

Example

Listings 21.137 and 21.138 demonstrate how to make the text in a text box wrap.

Listing 21.137 Using the TextBox.Wrap Property (C#)

```
MyTextBox.Wrap = true;
```

Listing 21.138 Using the TextBox.Wrap Property (Visual Basic.NET)

```
MyTextBox.Wrap = true
```

The CheckBox Class

The CheckBox Web control renders a check box HTML form element to the browser. This section covers all properties of the CheckBox class that are not derived from the WebControl base class.

Table 21.9 Properties of the CheckBox Class

Item	Description
<code>AutoPostBack</code>	Causes the page to automatically submit the data to the server when the contents of the check box change
<code>Checked</code>	Specifies the width of the rendered text box, in characters
<code>Text</code>	Specifies text to appear next to the rendered check box
<code> TextAlign</code>	Specifies the alignment of the text specified in the <code>Text</code> property

CheckBox.AutoPostBack

Syntax

Boolean AutoPostBack

Description

If the `CheckBox.AutoPostBack` property is set to True, the form will be submitted back to the server when the check box value is changed.

Example

Listings 21.139 and 21.140 demonstrate how to make a form automatically perform a postback to the server when the user clicks on the check box.

Listing 21.139 Using the CheckBox.AutoPostBack Property (C#)

```
MyCheckBox.AutoPostBack = true;
```

Listing 21.140 Using the CheckBox.AutoPostBack Property (Visual Basic.NET)

```
MyCheckBox.AutoPostBack = true
```

CheckBox.Checked

Syntax

Boolean Checked

Description

If the CheckBox.Checked property is set to True, the rendered check box will be checked by default.

Example

Listings 21.141 and 21.142 demonstrate how to make a check box appear checked when the page loads.

Listing 21.141 Using the CheckBox.Checked Property (C#)

```
MyCheckBox.Checked = true;
```

Listing 21.142 Using the CheckBox.Checked Property (Visual Basic.NET)

```
MyCheckBox.Checked = true
```

CheckBox.Text

Syntax

String Text

Description

The CheckBox.Text property specifies any text that should appear alongside the check box.

Example

Listings 21.143 and 21.144 demonstrate how to specify text to appear near the check box.

Listing 21.143 Using the CheckBox.Text Property (C#)

```
MyCheckBox.Text = "Select this item";
```

Listing 21.144 Using the CheckBox.Text Property (Visual Basic.NET)

```
MyCheckBox.Text = "Select this item"
```

CheckBox.TextAlign

Syntax

```
 TextAlign TextAlign
```

Description

The CheckBox.TextAlign property specifies the alignment of text (specified in the CheckBox.Text property). Valid values are Left and Right.

Example

Listings 21.145 and 21.146 demonstrate how to align text to the left of the check box.

Listing 21.145 Using the CheckBox.TextAlign Property (C#)

```
MyCheckBox.TextAlign = TextAlign.Left;
```

Listing 21.146 Using the CheckBox.TextAlign Property (Visual Basic.NET)

```
MyCheckBox.TextAlign = TextAlign.Left
```

The ImageButton Class

The ImageButton Web control is used to display on the page an image that has the functionality of a button. The properties of the ImageButton class that are not derived from the WebControl base class are covered in this section.

Table 21.10 Properties of the ImageButton Class

Item	Description
AlternateText	Specifies any text that should appear if the browser does not support images
CommandArgument	Specifies additional information that is passed to the method handling the click event
CommandName	Specifies the name of the command that should be performed if the user clicks on the button
ImageAlign	Specifies how the rendered image should be aligned on the page
ImageUrl	Specifies the URL of the image that is used to display the image button

ImageButton.AlternateText

Syntax

String AlternateText

Description

The `ImageButton.AlternateText` property specifies any text that should appear if the browser does not support images or if the images have been disabled.

Example

Listings 21.147 and 21.148 demonstrate how to specify the alternate text for an image that is rendered by the `ImageButton` control.

Listing 21.147 Using the ImageButton.AlternateText Property (C#)

```
MyImageButton.AlternateText = "Click here to submit";
```

Listing 21.148 Using the ImageButton.AlternateText Property (Visual Basic.NET)

```
MyImageButton.AlternateText = "Click here to submit"
```

ImageButton.CommandArgument

Syntax

String CommandArgument

Description

The `ImageButton.CommandArgument` property specifies additional information that is passed to the method that is handling the click event.

Example

Listings 21.149 and 21.150 demonstrate how to use the `CommandArgument` property in conjunction with the `CommandName` property to pass useful information to the method that is processing the click event.

Listing 21.149 Using the ImageButton.CommandArgument Property (C#)

```
MyImageButton.CommandName = "sort";
MyImageButton.CommandArgument = "byDate";
```

Listing 21.150 Using the ImageButton.CommandArgument Property (Visual Basic.NET)

```
MyImageButton.CommandName = "sort"
MyImageButton.CommandArgument = "byDate"
```

ImageButton.CommandName

Syntax

String CommandName

Description

The `ImageButton.CommandName` property specifies the name of the command that should be performed if the user clicks on the button. It is passed as an argument to the event handler.

Example

Listings 21.151 and 21.152 demonstrate how to use the `CommandArgument` property in conjunction with the `CommandName` property to pass useful information to the method that is processing the click event.

Listing 21.151 Using the ImageButton.CommandName Property (C#)

```
MyImageButton.CommandName = "sort";
MyImageButton.CommandArgument = "byDate";
```

Listing 21.152 Using the ImageButton.CommandName Property (Visual Basic.NET)

```
MyImageButton.CommandName = "sort"
MyImageButton.CommandArgument = "byDate"
```

ImageButton.ImageAlign

Syntax

ImageAlign ImageAlign

Description

The `ImageButton.ImageAlign` property specifies how the rendered image should be aligned on the page. Valid values are `AbsBottom`, `AbsMiddle`, `Baseline`, `Bottom`, `Left`, `Middle`, `NotSet`, `Right`, `TextTop`, and `Top`.

Example

Listings 21.153 and 21.154 demonstrate how to align the image button to the center of its container.

Listing 21.153 Using the ImageButton.ImageAlign Property (C#)

```
MyImageButton.ImageAlign = ImageAlign.Middle;
```

Listing 21.154 Using the ImageButton.ImageAlign Property (Visual Basic.NET)

```
MyImageButton.ImageAlign = ImageAlign.Middle
```

ImageButton.ImageUrl

Syntax

```
String ImageUrl
```

Description

The `ImageButton.ImageUrl` property specifies the URL of the image that is used to display the image button.

Example

Listings 21.155 and 21.156 demonstrate how to specify the `ImageUrl` of an image button.

Listing 21.155 Using the ImageButton.ImageUrl Property (C#)

```
MyImageButton.ImageUrl = "/images/imageButton.gif";
```

Listing 21.156 Using the ImageButton.ImageUrl Property (Visual Basic.NET)

```
MyImageButton.ImageUrl = "/images/imageButton.gif"
```

The LinkButton Class

The `LinkButton` Web control is used to display on the page a link that has the functionality of a button. The properties of the `LinkButton` class that are not derived from the `WebControl` base class are covered in this section.

Table 21.11 Properties of the LinkButton Class

Item	Description
CommandArgument	Specifies additional information that is passed to the method handling the click event
CommandName	Specifies the name of the command that should be performed if the user clicks on the button
Text	Specifies the text to be displayed for the rendered hyperlink button

LinkButton.CommandArgument

Syntax

```
String CommandArgument
```

Description

The LinkButton.CommandArgument property specifies additional information that is passed to the method handling the click event.

Example

Listings 21.157 and 21.158 demonstrate how to use the CommandArgument property in conjunction with the CommandName property to pass useful information to the method that is processing the click event.

Listing 21.157 Using the LinkButton.CommandArgument Property (C#)

```
MyLinkButton.CommandName = "sort";
MyLinkButton.CommandArgument = "byDate";
```

Listing 21.158 Using the LinkButton.CommandArgument Property (Visual Basic.NET)

```
MyLinkButton.CommandName = "sort"
MyLinkButton.CommandArgument = "byDate"
```

LinkButton.CommandName

Syntax

String CommandName

Description

The LinkButton.CommandName property specifies the name of the command that should be performed if the user clicks on the button. It is passed as an argument to the event handler.

Example

Listings 21.159 and 21.160 demonstrate how to use the CommandArgument property in conjunction with the CommandName property to pass useful information to the method that is processing the click event.

Listing 21.159 Using the LinkButton.CommandName Property (C#)

```
MyLinkButton.CommandName = "sort";
MyLinkButton.CommandArgument = "byDate";
```

Listing 21.160 Using the LinkButton.CommandName Property (Visual Basic.NET)

```
MyLinkButton.CommandName = "sort"
MyLinkButton.CommandArgument = "byDate"
```

LinkButton.Text

Syntax

String Text

Description

The LinkButton.Text property specifies the text to be displayed for the rendered hyperlink button.

Example

Listings 21.161 and 21.162 demonstrate how to specify the text of a link button.

Listing 21.161 Using the LinkButton.Text Property (C#)

```
MyLinkButton.Text = "Click here to submit!" ;
```

Listing 21.162 Using the LinkButton.Text Property (Visual Basic.NET)

```
MyLinkButton.Text = "Click here to submit!"
```

The RadioButton Class

The RadioButton Web control is used to display a radio button on the Web form. The properties of the RadioButton class that are not derived from the WebControl base class are covered in this section.

Table 21.12 Properties of the RadioButton Class

Item	Description
AutoPostBack	Causes the form to be submitted back to the server if the value of the RadioButton is changed
Checked	Specifies the selected state of the rendered radio button
GroupName	Specifies the group to which the radio button belongs
Text	Specifies the text to be displayed next to the rendered radio button
TextAlign	Specifies how text should be aligned next to the radio button

RadioButton.AutoPostBack

Syntax

Boolean AutoPostBack

Description

If the `RadioButton.AutoPostBack` property is set to True, the form will be submitted back to the server if the check box value is changed.

Example

Listings 21.163 and 21.164 demonstrate how to make the form automatically perform a postback to the server if the user clicks on a radio button.

Listing 21.163 Using the RadioButton.AutoPostBack Property (C#)

```
MyRadioButton.AutoPostBack = true;
```

Listing 21.164 Using the RadioButton.AutoPostBack Property (Visual Basic.NET)

```
MyRadioButton.AutoPostBack = true
```

RadioButton.Checked

Syntax

Boolean Checked

Description

If the `RadioButton.Checked` property is set to True, the rendered radio button will be True by default.

Example

Listings 21.165 and 21.166 demonstrate how to check a radio button automatically.

Listing 21.165 Using the RadioButton.Checked Property (C#)

```
MyRadioButton.Checked = true;
```

Listing 21.166 Using the RadioButton.Checked Property (Visual Basic.NET)

```
MyRadioButton.Checked = true
```

RadioButton.GroupName

Syntax

String GroupName

Description

The `RadioButton.GroupName` property specifies the group to which the radio button belongs. Only one radio button in a group can be set to True at any one time.

Example

Listings 21.167 and 21.168 demonstrate how to set the group name of a radio button.

Listing 21.167 Using the RadioButton.GroupName Property (C#)

```
MyRadioButton.GroupName = "question1";
```

Listing 21.168 Using the RadioButton.GroupName Property (Visual Basic.NET)

```
MyRadioButton.GroupName = "question1"
```

RadioButton.Text

Syntax

String Text

Description

The RadioButton.Text property specifies text that should appear next to the radio button.

Example

Listings 21.169 and 21.170 demonstrate how to specify the text next to a radio button.

Listing 21.169 Using the RadioButton.Text Property (C#)

```
MyRadioButton.Text = "option1";
```

Listing 21.170 Using the RadioButton.Text Property (Visual Basic.NET)

```
MyRadioButton.Text = "option1"
```

RadioButton.TextAlign

Syntax

TextAlign TextAlign

Description

The RadioButton.TextAlign property specifies how text (specified in the RadioButton.TextAlign property) should be aligned within its container. For instance, if the RadioButton is contained with a panel, the TextAlign property controls the RadioButton's alignment within that panel. Valid values are Left and Right.

Example

Listings 21.171 and 21.172 demonstrate how to specify the alignment of the radio button's text.

Listing 21.171 Using the RadioButton.TextAlign Property (C#)

```
MyRadioButton.TextAlign = TextAlign.Left;
```

Listing 21.172 Using the RadioButton.TextAlign Property (Visual Basic.NET)

```
MyRadioButton.TextAlign = TextAlign.Left
```

The BaseValidator Class

In addition to deriving properties from the `WebControl` base class, validation controls also derive from the `BaseValidator` class. The `BaseValidator` class includes properties that are common to all validation controls.

Table 21.13 Properties of the BaseValidator Class

Item	Description
<code>ControlToValidate</code>	Specifies the ID of a control on which the validation control is to perform validation
<code>Display</code>	Specifies how the validation control should appear on the page
<code>EnableClientScript</code>	Specifies whether client-side script should be enabled or disabled
<code>ErrorMessage</code>	Contains the text that is displayed if the validation control reports an error
<code>IsValid</code>	Contains true if the user's input meets the validation requirements
<code>Text</code>	Specifies alternate text to be displayed if validation fails

BaseValidator.ControlToValidate

Syntax

```
String ControlToValidate
```

Description

The `BaseValidator.ControlToValidate` property specifies the ID of a control on which the validation control is performing validation.

Example

Listings 21.173 and 21.174 demonstrate how to specify which control the validation control should validate. `RequiredFieldValidator` is used for the purposes of this example, but any validation control would be set in the same way.

Listing 21.173 Using the BaseValidator.ControlToValidate Property (C#)

```
MyRequiredFieldValidator.ControlToValidate = "TargetControlID";
```

Listing 21.174 Using the BaseValidator.ControlToValidate Property (Visual Basic.NET)

```
MyRequiredFieldValidator.ControlToValidate = "TargetControlID"
```

BaseValidator.Display

Syntax

ValidatorDisplay Display

Description

The `BaseValidator.Display` property specifies how the validation control should appear on the page. If it is set to `Dynamic`, the validation control will not take up space until an error message is generated. If it is set to `Static`, the validation control will always take up a consistent amount of space. If it is set to `None`, the validation control will not appear on the page.

Example

Listings 21.175 and 21.175 demonstrate how to hide a validation control.

Listing 21.175 Using the BaseValidator.Display Property (C#)

```
MyRequiredFieldValidator.Display = ValidatorDisplay.None;
```

Listing 21.176 Using the BaseValidator.Display Property (Visual Basic.NET)

```
MyRequiredFieldValidator.Display = ValidatorDisplay.None
```

BaseValidator.EnableClientScript

Syntax

Boolean EnableClientScript

Description

The `BaseValidator.EnableClientScript` property specifies whether client-side script should be enabled or disabled.

Example

Listings 21.177 and 21.178 demonstrate how to turn off client-side script for a validation control.

Listing 21.177 Using the BaseValidator.EnableClientScript Property (C#)

```
MyRequiredFieldValidator.EnableClientScript = false;
```

***Listing 21.178 Using the BaseValidator.EnableClientScript Property
(Visual Basic.NET)***

```
MyRequiredFieldValidator.EnableClientScript = false
```

BaseValidator.ErrorMessage

Syntax

```
String ErrorMessage
```

Description

The `BaseValidator(ErrorMessage)` contains the text that is displayed if the validation control reports an error.

Example

Listings 21.179 and 21.180 demonstrate how to specify the error message that will appear if validation of a control fails.

Listing 21.179 Using the BaseValidator.ErrorMessage Property (C#)

```
MyRequiredFieldValidator.ErrorMessage = "Enter required Text";
```

***Listing 21.180 Using the BaseValidator.ErrorMessage Property (Visual
Basic.NET)***

```
MyRequiredFieldValidator.ErrorMessage = "Enter required Text"
```

BaseValidator.IsValid

Syntax

```
Boolean IsValid
```

Description

The `BaseValidator.IsValid` property contains `True` if the user's input meets the validation requirements of the control. Otherwise, it contains `False`.

Example

Listings 21.181 and 21.182 demonstrate how to retrieve the validation state of a validation control.

Listing 21.181 Using the BaseValidator.IsValid Property (C#)

```
MyRequiredFieldValidator.IsValid;
```

Listing 21.182 Using the BaseValidator.IsValid Property (Visual Basic.NET)

```
MyRequiredFieldValidator.IsValid
```

BaseValidator.Text

Syntax

String Text

Description

The BaseValidator.Text property enables alternate text to be displayed if validation fails. If the Text property is specified, the alternate text will be displayed on the page, regardless of whatever it is specified in the ErrorMessage property. Similarly, the ErrorMessage property is always displayed in the ValidationSummary control.

Example

Listings 21.183 and 21.184 demonstrate how to specify the text of a validation control.

Listing 21.183 Using the BaseValidator.Text Property (C#)

```
MyRequiredFieldValidator.Text = "*";
```

Listing 21.184 Using the BaseValidator.Text Property (Visual Basic.NET)

```
MyRequiredFieldValidator.Text = "*"
```

The CompareValidator Class

The CompareValidator Web control is used to compare the value of one Web control to another Web control or to a static value.

Table 21.14 Properties of the CompareValidator Class

Item	Description
ControlToCompare	Specifies which control the control specified in the ControlToValidate property is being compared against
Operator	Specifies which comparison operator is being used in order to determine validation status
ValueToCompare	Specifies a value to use in comparing the two controls

CompareValidator.ControlToCompare

Syntax

String ControlToValidate

Description

The `CompareValidator.ControlToCompare` property specifies which control the control specified in the `ControlToValidate` property is being compared against.

Example

Listings 21.185 and 21.186 demonstrate how to specify the `ControlToCompare` property of a compare validator.

Listing 21.185 Using the CompareValidator.ControlToCompare Property (C#)

```
MyCompareValidator.ControlToCompare = "TargetControlID";
```

Listing 21.186 Using the CompareValidator.ControlToCompare Property (Visual Basic.NET)

```
MyCompareValidator.ControlToCompare = "TargetControlID"
```

CompareValidator.Operator

Syntax

ValidationCompareOperator Operator

Description

The `CompareValidator.Operator` property specifies which comparison operator is being used in order to determine validation status. Proper values are `DataTypeCheck`, `Equal`, `GreaterThan`, `GreaterThanOrEqualTo`, `LessThan`, `LessThanOrEqualTo`, and `NotEqual`.

Example

Listings 21.187 and 21.188 demonstrate how to specify the operator to use for comparing two controls.

Listing 21.187 Using the CompareValidator.Operator Property (C#)

```
MyCompareValidator.Operator = ValidationCompareOperator.Equal;
```

Listing 21.188 Using the CompareValidator.Operator Property (Visual Basic.NET)

```
MyCompareValidator.Operator = ValidationCompareOperator.Equal
```

CompareValidator.ValueToCompare

Syntax

String ValueToCompare

Description

The CompareValidator.ValueToCompare property enables you to specify a specific value to compare the value in the ControlToValidate property against.

Example

Listings 21.189 and 21.190 demonstrate how to specify the static value to compare to a control.

Listing 21.189 Using the CompareValidator.ValueToCompare Property (C#)

```
MyCompareValidator.ValueToCompare = "12345";
```

Listing 21.190 Using the CompareValidator.ValueToCompare Property (Visual Basic.NET)

```
MyCompareValidator.ValueToCompare = "12345"
```

The CustomValidator Class

The CustomValidator validation control is used to perform validation that other validation controls either can't easily do or can't do at all. Performing advanced credit card validation is a good instance for the need for a CustomValidator validation control.

Table 21.15 Properties of the CustomValidator Class

Item	Description
ClientValidationFunction	Specifies which client-side function will be used to perform the custom client-side validation

CustomValidator.ClientValidationFunction

Syntax

```
String ClientValidationFunction
```

Description

The CustomValidator.ClientValidationFunction property specifies which client-side function will be used to perform the custom client-side validation.

Example

Listings 21.191 and 21.192 demonstrate how to specify the client validation function used to perform custom validation on the client side.

Listing 21.191 Using the CustomValidator.ClientValidationFunction Property (C#)

```
MyCustomValidator.ClientValidationFunction = "ValidateItems";
```

Listing 21.192 Using the CustomValidator.ClientValidationFunction Property (Visual Basic.NET)

```
MyCustomValidator.ClientValidationFunction = "ValidateItems"
```

The RangeValidator Class

The RangeValidator validation control makes sure that the target Web control is between two values, denoted by the MinimumValue and MaximumValue.

Table 21.16 Properties of the RangeValidator Class

Item	Description
MaximumValue	Specifies the maximum value that the validated field can contain
MinimumValue	Specifies the minimum value that the validated field can contain

RangeValidator.MaximumValue

Syntax

```
String MaximumValue
```

Description

The RangeValidator.MaximumValue property specifies the maximum value that the validated field can contain.

Example

Listings 21.193 and 21.194 demonstrate how to specify the maximum value for a RangeValidator control.

Listing 21.193 Using the RangeValidator.MaximumValue Property (C#)

```
MyRangeValidator.MaximumValue = "10";
```

Listing 21.194 Using the RangeValidator.MaximumValue Property (Visual Basic.NET)

```
MyRangeValidator.MaximumValue = "10"
```

RangeValidator.MinimumValue

Syntax

```
String MinimumValue
```

Description

The RangeValidator.MinimumValue property specifies the minimum value that the validated field can contain.

Example

Listings 21.195 and 21.196 demonstrate how to specify the minimum value of a RangeValidator control.

Listing 21.195 Using the RangeValidator.MinimumValue Property (C#)

```
MyRangeValidator.MinimumValue = "1";
```

Listing 21.196 Using the RangeValidator.MinimumValue Property (Visual Basic.NET)

```
MyRangeValidator.MinimumValue = "1"
```

The RegularExpressionValidator Class

The RegularExpressionValidator validation control is powerful; it enables you to perform validation by using the powerful Perl-based regular expression language. This is primarily used to make sure user input is in a certain format or contains certain values.

Table 21.17 Properties of the RegularExpressionValidator Class

Item	Description
ValidationExpression	Specifies the regular expression that is used to determine validation status

RegularExpressionValidator.ValidationExpression

Syntax

```
String ValidationExpression
```

Description

The RegularExpressionValidator.ValidationExpression property specifies the Perl-based regular expression that is used to determine validation on the value in the ControlToValidate property.

Example

Listings 21.197 and 21.198 demonstrate how to specify the validation expression for a RegularExpression validation control.

Listing 21.197 Using the ActiveXControl Property (C#)

```
MyRegularExprValidator.ValidationExpression = "\d{5}";
```

Listing 21.198 Using the ActiveXControl Property (Visual Basic.NET)

```
MyRegularExprValidator.ValidationExpression = "^\\d{5}$"
```

The RequiredFieldValidator Class

The RequiredFieldValidator validation control checks to make sure that the user has entered something into the field on which validation is being performed.

Table 21.19 Properties of the RequiredFieldValidator Class

Item	Description
ValueToCompare	Contains the initial value present in the target control

RequiredFieldValidator.InitialValue

Syntax

```
String initialValue
```

Description

The RequiredFieldValidator.InitialValue property contains the initial value present in the target control. The target control will pass validation as long as it isn't equal to this initial value. This property is often used when you want to suggest an input format to the user, such as when entering a phone number or date.

Example

Listings 21.199 and 21.200 demonstrate how to set the InitialValue property of a RequiredFieldValidator control.

Listing 21.199 Using the RequiredFieldValidator.InitialValue Property (C#)

```
MyRequiredFieldValidator.InitialValue = "MM-DD-YYYY";
```

Listing 21.200 Using the RequiredFieldValidator.InitialValue Property (Visual Basic.NET)

```
MyRequiredFieldValidator.InitialValue = "MM-DD-YYYY"
```

The ValidationSummary Class

The ValidationSummary validation control is used to display all validation errors in a single place on the page.

Table 21.20 Properties of the ValidationSummary Class

Item	Description
DisplayMode	Specifies how the validation summary control appears on the page
HeaderText	Specifies the text that appears at the top of the rendered control on the page
ShowMessageBox	Specifies whether the validation summary is displayed in a message box or on the page
ShowSummary	Specifies whether the validation summary is displayed on the page
MaximumValue	Specifies the maximum value that the validated field can contain
MinimumValue	Specifies the minimum value that the validated field can contain

ValidationSummary.DisplayMode

Syntax

```
ValidationSummaryDisplayMode DisplayMode
```

Description

The ValidationSummary.DisplayMode property specifies how the validation summary control will appear on the page. Valid values are BulletList, List, and SingleParagraph.

Example

Listings 21.201 and 21.202 demonstrate how to display the ValidationSummary control on the page in SingleParagraph format.

Listing 21.201 Using the ValidationSummary.DisplayMode Property (C#)

```
MyValidationSummary.DisplayMode = ValidationSummaryDisplayMode.SingleParagraph;
```

Listing 21.202 Using the ValidationSummary.DisplayMode Property (Visual Basic.NET)

```
MyValidationSummary.DisplayMode = ValidationSummaryDisplayMode.SingleParagraph;
```

ValidationSummary.HeaderText

Syntax

```
String HeaderText
```

Description

The ValidationSummary.HeaderText property specifies the text that appears at the top of the ValidationSummary control.

Example

Listings 21.203 and 21.204 demonstrate how to specify the header text of a validation control.

Listing 21.203 Using the ValidationSummary.HeaderText Property (C#)

```
MyValidationSummary.HeaderText = "Please correct the following fields";
```

Listing 21.204 Using the ValidationSummary.HeaderText Property (Visual Basic.NET)

```
MyValidationSummary.HeaderText = "Please correct the following fields"
```

ValidationSummary.ShowMessageBox

Syntax

```
Boolean ShowMessageBox
```

Description

The ValidationSummary.ShowMessageBox property specifies whether the validation summary is displayed in a message box or on the page.

Example

Listings 21.205 and 21.206 demonstrate how to display validation summary results in a message box.

Listing 21.205 Using the ValidationSummary.ShowMessageBox Property (C#)

```
MyValidationSummary.ShowMessageBox = true;
```

Listing 21.206 Using the ValidationSummary.ShowMessageBox Property (Visual Basic.NET)

```
MyValidationSummary.ShowMessageBox = true
```

ValidationSummary>ShowSummary

Syntax

```
Boolean ShowSummary
```

Description

The ValidationSummary.ShowSummary property specifies whether the validation summary should be displayed on the page.

Example

Listings 21.207 and 21.208 demonstrate how to turn off the validation summary.

Listing 21.207 Using the ValidationSummary.ShowSummary Property (C#)

```
MyValidationSummary.ShowSummary=false;
```

Listing 21.208 Using the ValidationSummary.ShowSummary Property (Visual Basic.NET)

```
MyValidationSummary.ShowSummary=false
```

Selected Static Classes

Some classes in the `System.Web.UI.WebControls` namespace exist mainly to provide standard lookup values for commonly accessed information. These classes are listed here as a quick reference.

The FontInfo Class

The following are the properties of the `FontInfo` class:

```
Bold  
Italic  
Name  
Names  
Overline  
Size  
Strikeout  
Underline
```

The following are the properties of the `FontSize` class:

```
AsUnit  
Large  
Larger  
Medium  
NotSet  
Small  
Smaller  
XLarge  
XSmall  
XXLarge  
XXSmall
```

The ImageAlign Class

The following are the properties of the `ImageAlign` class:

```
AbsBottom  
AbsMiddle  
Baseline  
Bottom  
Left  
Middle
```

NotSet
Right
TextTop
Top

The TextAlign Class

The following are the properties of the TextAlign class:

Left
Right

The BorderStyle Class

The following are the properties of the BorderStyle class:

Dashed
Dotted
Double
Groove
Inset
None
NotSet
Outset
Ridge
Solid

The ValidationSummaryDisplayMode Class

The following are the properties of the ValidationSummaryDisplayMode class:

BulletList
List
SingleParagraph



INDEX

A

- aborting transactions, 259
- AbsBottom property, 580
- AbsMiddle property, 580
- abstraction, components, 244
- AcceptTypes property, 488-489
- accessing
 - class members, 252
 - components, 253-255
 - DataReader object, 218-219
 - keys, 523-524
- Accounts object, web.config file, 267
- AccountsDebit stored procedure, 261
- ActiveXControls property, 469-471
- Adapter() method
 - ArrayList class, 305, 310, 311
 - Hashtable class, 337, 341
 - SortedListed class, 355, 361
 - SqlParameterCollection class, 453, 456-457
- AddFileDependency() method, 508
- adding
 - borders, 82
 - cookies, 508, 516
 - files, 508
 - headers, 508, 516
 - objects
 - to arraylists, 305, 311, 317-318
 - to hashtables, 337, 341
 - to queues, 347, 353
 - to sortedlists, 355, 361
 - to stacks, 373
 - parameters, 453, 456-457, 461-462
 - user controls, 114-116
 - Web controls, 36
- see also* inserting

AddOnCheckedChanged property, 88, 93
AddOnClick() method, 84
AddOnSelectedIndexChanged Datasource property, 99
AddOnSelectedIndexChanged property, 106
AddOnTextChanged, 96
AddRange() method, 305, 311-312
addresses, users, 489, 505
ADO.NET, 55-58
ADOCommand, 120, 212
ADOConnection, 58, 120, 211
ADODataAdapter, 120
ADODataReader, 120
AdRotator control, 38
agents, users, 489, 504
alignment, horizontal/vertical, 144, 158
AllowingSorting property, 175
AllowPaging property, 171
AlternateText property, 552, 561-562
AlternatingItemStyle property, 143, 157
AlternatingItemTemplate template, 135, 139
And() method, 327, 330-331
AOL property, 469, 471
AppendCookie() method, 508, 516
AppendHeader() method, 508, 516
AppendToLog() method, 508, 517
ApplicationPath property, 488, 490
applications
 ASP.NET, 296-299
 caching, 291-292
 COM+, 259
 components, 253-255
 configuration, 287-288
 deploying, 295
 errors, 289-290
 root directory, 488, 500
 virtual paths, 488, 493
arguments, 279-280
ArrayList class, 120
 methods
 Adapter(), 305, 310
 Add(), 305, 311
 AddRange(), 305, 311-312
 BinarySearch(), 305, 312
 Clear(), 305, 313
 Clone(), 305, 313-314
 Contains(), 305, 314
 CopyTo(), 305, 314-315
 FixedSize(), 305, 315-316
 GetRange(), 305, 316
 IndexOf(), 305, 317
 Insert(), 305, 317-318
 InsertRange(), 305, 318-319
 LastIndexOf(), 305, 319
 ReadOnly(), 305, 320
 Remove(), 305, 320-321
 RemoveAt(), 305, 321
 RemoveRange(), 305, 321-322
 Repeat(), 305, 322
 Reverse(), 305, 323
 SetRange(), 305, 323-324
 Sort(), 305, 324-325
 Synchronized(), 305, 325
 ToArray(), 305, 326
 TrimToSize, 305, 326
 properties, 305-310
 System.Collections namespace, 304-309
arraylists
 converting, 305, 326
 copying, 305, 314-315
 counting, 305, 307
 duplicating, 305, 313-314
 indexes, 305, 309-310, 317, 319
 length, 305-306
 list controls, 122-125
 objects, 305, 311, 317-321
 ranges, 305, 311-312, 316, 318-319, 321-322
 read-only, 305, 308, 320
 repeating, 305, 322
 returning, 305, 310
 reversing, 305, 323
 searches, 305, 312
 SetRange, 305, 323-324
 sizing, 305, 307-308, 315-316, 326
 sorting, 305, 324-325
 synchronization, 305, 309, 325
 True value, 305, 308
 values, 305, 314
arrays, 347, 354-355, 373, 377-378, 380
ascx file extension, 111
ASP.NET
 applications, 291-292
 basic authentication, 297
 config.web file, 298
 controls, 7
 designing, 296
 form-based authentication, 298

- installation, 294
- Microsoft .NET support, 5
- networks, 297
- NTLM authentication, 298
- operating systems, 297
- pages, 299
- proxy classes, 281
- user controls, 111-117
- URLs, 298
- versions, 5
- Web Forms, 7, 21, 28, 29, 36, 284
- assemblies**, *see* components
- assigning Command object**, 221-223
- AsUnit property**, 580
- attributes, runat="server,"** 31
- Attributes property**, 523, 525
- authentication**
 - basic, 297
 - config.web files, 298
 - form-based, 298
 - login URLs, 298
 - NTLM, 298
 - pages, 299
 - users, 488, 498
- AutoComplete**, 260
- AutoGenerateColumns property**, 154
- automating**
 - columns, 43, 154
 - configuration changes detection, 11
 - Intermediate Language, 23-26
 - transactions, 260
 - ViewState management, 34
- AutoPostBack property**
 - CheckBox class, 559
 - RadioButton class, 566
 - TextBox class, 555
- B**
 - BackColor property**, 144, 158, 523, 525
 - BackgroundSounds property**, 469, 471
 - BackImageUrl property**, 553
 - base class**, Web controls, 37
 - Baseline property**, 580
 - BaseValidator class properties**, 569-572
 - basic authentication**, ASP.NET application security, 297
 - BeginTransaction() method**, 238, 395, 399-400
 - benefits, assemblies**, 15
- Beta property**, 469, 472
- binary characters**, 508, 517-518
- Binary property**, 448
- BinarySearch() method**, 305, 312
- BinaryWrite() method**, 508, 517-518
- binding**
 - data sources, 47-48
 - expressions, 121-122
 - list controls, 122-128, 129-134
- Bit property**, 448
- bitarrays**
 - And operation, 327, 330-331
 - classes, 326, 328-336
 - copying, 327, 331-332
 - indexes, 327, 329
 - length, 327, 330
 - location, 327, 332-333, 335
 - Not operation, 327
 - objects, 327-328
 - Or operation, 327
 - read-only, 327-328
 - synchronization, 327-329
 - values, 327, 335-336
- BitInt property**, 448
- bold attribute**, 144, 158
- Bold property**, 580
- bool AutoPostBack**
 - CheckBox control, 88
 - Datasource property, 99
 - ListBox control, 106
 - RadioButton control, 93
 - TextBox control, 96
- bool Checked property**, 88, 93
- Bool wrap property**, 76, 96
- Boolean IsValid property**, 187
- Boolean OnServerValidate property**, 201
- Boolean ShowMessageBox property**, 205
- Boolean ShowSummary property**, 205
- Boolean values**, 403, 409-410
- BorderColor property**, 144, 158, 523, 525
- BorderStyle class**, 82, 144, 158, 523, 526, 581
- BorderWidth property**, 144, 158, 523, 526
- Bottom property**, 580
- BoundColumn column type properties**, 162-163
- Browser property**, 469, 472, 488, 490
- browsers**
 - downlevel, 50
 - list controls, 43
 - requests, 488, 490

types, 470, 478-479

uplevel, 50

versions, 470, 479

Web controls, 37

Buffer property, 508-509

BufferOutput property, 508-510

buffers, 508-510, 518

BulletList property, 581

business logic layer (BLL), 244

Button class, 84-87, 535-536

ButtonColumn column type, 162, 165

ButtonType property, 165, 166

Byte value, 403, 410-412

bytes, 488, 503

C

C# programming language, 452

Accounts object, 265-267

ActiveXControls, 470, 576

AlternateText property, 552, 562

AOL, 471

applications, 490, 500

arraylists

 copying, 315

 counting, 307

 creating, 326

 duplicating, 313

 indexes, 310, 317, 319

 length, 306

 objects, 311, 313, 318, 321

 ranges, 311, 316, 318, 322, 324

 read-only, 308, 320

 repeating, 322

 returning, 310

 reversing, 323

 searches, 312

 sizing, 307, 316

 sorting, 325

 synchronization, 309, 325

 trimming, 326

 True value, 314

AutoPostBack property, 556, 560, 567

BackImageUrl property, 554

beta versions, 472

bitarrays, 327-329, 331, 333-336

browsers, 472, 476, 478-479, 490

buffers, 509, 518, 520

Button control, 85

bytes, 503

cache, 510, 513

CDF, 473

CellPadding property, 538

CellSpacing property, 539

CheckBox control, 88-89

CheckBoxList control, 90-91

Checked property, 560, 567

Clear() method, 375

clients, 491, 521

ClientValidationFunction property, 574

Clone() method, 376

collections, 454

columns, 406, 556

Command object, 221-222

command objects, 401

CommandArgument property, 536, 562, 565

CommandName property, 536, 563, 565

CommandText property, 386

CommandTimeout property, 386

CommandType property, 387

COMP proxy class, 257

Compare validation control, 196-198

components, 253, 254

Connection object, 212

connections, 387, 498, 519

ConnectionString property, 395

ConnectionTimeout property, 396

Contains() method, 377

ControlToCompare property, 573

ControlToValidate property, 570

cookies, 473, 482-487, 512, 516

CopyTo() method, 377

crawler search engines, 474

CreateParameter() method, 391

Custom validation control, 202-204

data, 521

data transfer methods, 497

DataAdapter object, 220

Database property, 397

databases, 400-402

DataGridView control, 155

 applying styles, 159-160

 objects, 166-168

 paging, 171-173

 sorting, 176-177

DownList control, 140-141, 145-146,

 151-152

DataReader object, 215-216, 218-219

datasets, 226

DataSource property, 397
 DayHeaderStyle property, 539
 DayNameFormat property, 540
 DayStyle property, 540
 Display property, 570
 DisplayMode property, 578
 DropDownList control, 100, 102-103
 ECMA script, 474
 EnableClientScript property, 571
 ErrorMessage property, 571
 ExecuteNonQuery() method, 214, 391
 ExecuteReader() method, 392
 ExecuteScalar() method, 213, 393
 ExecuteXmlReader() method, 394
 FieldCount property, 404
 fields, 247, 414, 419, 425, 426
 files, 493-495, 522
 FirstDayOfWeek property, 541
 Float values, 420
 forms, 497
 frames, 475
 GetBoolean() values, 409
 GetByte() value, 410
 GetBytes() value, 411
 GetChar() value, 412
 GetChars() value, 413
 GetDateTime() value, 415
 GetDecimal() value, 416
 GetDouble(), 418
 GroupName property, 568
 GUID value, 421
 hashtables
 copying, 345
 duplicating, 342
 keys, 340, 343-344
 objects, 337, 339, 341-342, 346
 read-only, 338
 synchronization, 346
 values, 341, 344
 headers, 497, 515-516, 519
 HeaderText property, 579
 HorizontalAlign property, 554
 HTML content, 515
 HTTP, 510, 511, 514-515
 ID property, 530
 IIS log file, 517
 Image control, 73-74
 ImageAlign property, 552, 563
 ImageButton control, 85
 ImageUrl property, 533, 553, 564
 InitialValue property, 577
 Int value, 422-424
 IsClosed property, 405
 IsValid property, 571
 Java applets, 475
 JavaScript, 475
 Label control, 69
 LinkButton control, 85
 list controls, 122-123, 126-127, 129,
 132-133
 ListBox control, 106-107
 Master/Detail view, 178-181
 MaximumValue property, 575
 MaxLength property, 556
 members, 250-251
 methods, 248
 MIME, 489, 492
 MinimumValue property, 576
 namespaces, 246
 NavigateUrl property, 534
 next results, 446
 NextMonthText property, 541
 NextPrevFormat property, 542
 NextPrevStyle property, 542
 Null value, 445
 Operator property, 573
 OtherMonthDayStyle property, 543
 output, 510, 517, 520
 PacketSize property, 398
 Page property, 530
 pages, 512
 Panel control, 76-77
 parameters, 449, 455, 499
 adding, 456, 461
 checking, 458
 deleting, 457, 462-463
 input, 228-229, 232-235
 input/output, 450
 location, 460
 names, 451
 null, 450
 output, 232-235
 precision, 451
 property, 388
 scale, 452
 values, 453
 Parent property, 531
 paths, 500
 Peek() method, 378
 physical disks, 501

Pics() method, 521
platforms, 477
Pop() method, 379
PrevMonthText property, 543
properties, 249
Push() method, 379
queries, 390, 501
queues, 348-355
RadioButton control, 94
Range validation control, 193-194
ReadOnly property, 557
records, 447
RecordsAffected property, 407
references, 256
RegularExpression validation control, 198-200
Repeater control, 137-138
requests, 491, 492, 502, 507
RequiredField validation control, 189-191
ResetCommandTimeout() method, 394
Rows property, 557
schema tables, 427
SelectedDate property, 544
SelectedDayStyle property, 544
SelectionMode property, 545
SelectMonthText property, 545
SelectorStyle property, 546
SelectWeekText property, 546
servers, 503
ServerVersion property, 398
ShowDayHeader property, 546
ShowGridLines property, 547
ShowMessageBox property, 579
ShowNextPrevMonth property, 547
ShowSummary property, 580
ShowTitle property, 548
16-bit platforms, 480
sortedlists
 adding, 361
 capacity, 356
 copying, 365
 counting, 357
 deleting, 361, 370
 duplicating, 362
 indexes, 366, 368-369, 371
 keys, 360, 363-364, 366-367
 objects, 359
 read-only, 358
 sizing, 372
synchronization, 358, 372
values, 360, 363-364, 368
sounds, 471
SqlBinary value, 428
SqlBit value, 429
SqlByte value, 430
SqlDataReader object, 408
SqlDateTime value, 431
SqlDecimal value, 432
SqlDouble value, 433
SqlGuid value, 434
SqlInt16 value, 435
SqlInt32 value, 436
SqlInt64 value, 437
SqlMoney value, 438
SqlSingle value, 439
stacks, 374, 375
State property, 399
stored procedures, 224
String value, 443
Synchronized() method, 380
System.Data.SqlClient namespace, 383-384
TabIndex property, 531
Table control, 79-80
tables, 478
Target property, 534
Text property, 535, 537, 551, 558, 560, 566, 568, 572
TextAlign property, 561, 569
TextBox control, 97-98
TextMode property, 558
32-bit platforms, 481
TitleFormat property, 548
TitleStyle property, 549
ToArray() method, 381
TodayDayStyle property, 549
TodaysDate property, 550
ToolTip property, 532
transactions, 258, 389
 aborting, 259
 beginning, 239
 committing, 259, 466
 isolation level, 465
 rollbacks, 467-468
 rolling back, 239
 saving, 240
 starting, 400
 support, 262-264
URLs, 502, 504

users, 114, 498, 505, 506
ValidationSummary control, 206-208
values, 444
ValueToCompare property, 574
VBScript, 479
virtual paths, 499
Visible property, 532
VisibleDate property, 550
W3C XML, 480
Web controls, 524-530
Web servers, 513
Web services, 61, 270-271, 273
WeekendDayStyle property, 551
Width property, 533
WorkstationId property, 399
Wrap property, 555, 559
XML documents, 477

caching
applications, 291
data, 12
expiration dates, 508, 512-513
fragment, 11-12
headers, 508, 510
output, 11
user controls, 117

calculations, 60

Calendar class properties
CellPadding, 537-538
CellSpacing, 537, 539
DayHeaderStyle, 537, 539
DayNameFormat, 537, 540
DayStyle, 537, 540
FirstDayOfWeek, 537, 541
NextMonthText, 537, 541
NextPrevFormat, 537, 542
NextPrevStyle, 537, 542
OtherMonthDayStyle, 537, 542
PrevMonthText, 537, 543
SelectedDate, 537, 543
SelectedDayStyle, 538, 544
SelectionMode, 538, 544
SelectMonthText, 538, 545
SelectorStyle, 538, 545
SelectWeekText, 538, 546
ShowDayHeader, 538, 546
ShowGridLines, 538, 547
ShowNextPrevMonth, 538, 547
ShowTitle, 538, 548
TitleFormat, 538, 548
TitleStyle, 538, 548

TodayDayStyle, 538, 549
TodaysDate, 538, 549
VisibleDate, 538, 550
WeekendDayStyle, 550

Calendar control, 38

calling Web services, 64

Cancel() method, 385, 390

canceling queries, 385, 390

CancelText property, 166

Capacity property, 305-306, 355-356

cascading style sheets (CSS), 52

casting, 121

CDF property, 469, 472-473

CellPadding property, 537-538

cells, 159

CellSpacing property, 537, 539

CERT Coordination Center, 297

certificates, 488, 490-491

ChangeDatabase() methods, 395, 400

changing databases, 395, 400

Channel Definition Format (CDF), 469, 472-473

Char property, 448

Char value, 403, 412-414

character sets, 488, 491, 508, 510

Charset property, 508, 510

CheckBox control, 88-93, 559-561

CheckBoxList control, 90-93

Checked property, 559-560, 566-567

checking parameters, 453, 458-459

classes, 247
members, 247-252
namespaces, 245
precompiled, 282
proxy, 278-281
serviced components, 258
SqlCommand, 385
System.Data.OleDbCommand, 212
System.Data.SqlClient, 211, 212
System.Data.SqlClient, 212
transactions, 259
Web services, 61, 270

Clear() method
ArrayList class, 305, 313
Hashtable class, 337, 341-342
HttpResponse class, 508, 518
Queue class, 347, 349-350
SortedListed class, 355, 361-362
SqlParameterCollection class, 453, 457
Stack class, 373, 375-376

ClearContent() method, 508, 518

ClearHeaders() method, 508, 519

clients

- certificates, 488, 490-491
- connections, 509, 519
- identification, 523, 527
- output, 509, 520
- scripts, 21, 50
- URLs, 509, 521
- validation, 201, 574

Clone() method

- ArrayList class, 305, 313-314
- BitArray class, 327, 331-332
- Hashtable class, 337, 342-343
- Queue class, 347, 350-351
- SortedList class, 355, 362
- Stack class, 373, 376

Close() method

- HttpResponse class, 509, 519
- SqlConnection class, 395, 401
- SqlDataReader class, 403, 408

closing

- clients, 509, 519
- databases, 395, 401
- SqlDataReader object, 403, 408

code

- inserting, 27
- Microsoft .NET framework, 9
- server-side, 21-26
- Web forms, 21
- Web services, 8

code behind, 29

collections

- indexes, 453, 455-456
- parameters, 488, 499

color, 144, 158

columns

- buttons, 165-166
- commands, 165
- configuration, 161-166
- DataList control, 148-149
- displaying, 163-166
- fields, 162, 165
- footers, 162-166
- formatting, 165
- generating, 154
- headers, 162-166
- read-only, 163
- sorting, 163-166
- styles, 165-166

text, 165-166, 555-556

types, 162-166

URLs, 164

COM+, 258-267

Command class, 58

Command object, 213, 221-223

command-line options, 253

CommandArgument property

- Button class, 535
- ImageButton class, 561-562
- LinkButton class, 564

CommandName property

- Button class, 535-536
- ButtonColumn column type, 165
- ImageButton class, 561, 563
- LinkButton class, 564-565

commands

- columns, 165
- data sources, 58
- executing, 385-387
- returning, 395, 401-402

CommandText property, 385-386

CommandTimeout property, 385-386

CommandType property, 385-387

committing transactions, 241, 259, 465-466

Common Language Runtime engine, 6

Compare validation control, 195-198

CompareValidator class, 52, 572-573

compiling, 10

- command-line options, 253
- components, 252-253
- JIT compilation, 14
- just-in-time, 282
- Microsoft .NET language, 14
- troubleshooting, 22
- Web forms, 22-26

components

- abstraction, 244
- accessing, 253-255
- assemblies, 243
- benefits, 15
- compiling, 252-253
- delegation, 244
- deploying, 295-296
- editing, 15
- encapsulation, 252
- IL Disassembler, 15
- implementation, 244
- maintenance, 244
- manifests, 14

- registration, 15
- serviced, 258
- sharing, 15
- configuration**
 - applications, 288
 - authentication, 298
 - DataGridView control, 161-166
 - ListBox control, 142, 144-148
 - detection, 11
 - error messages, 188-189
 - files, 287, 288
 - Microsoft .NET framework, 11
- Connection object, 58, 212-213, 238**
- connections**
 - clients, 509, 519
 - data sources, 211-212
 - databases, 385, 387, 395-397
 - secure, 488, 498
 - state, 395, 398-399
 - strings, 395-396
 - timeouts, 395-396
 - Web servers, 508, 513
- ConnectionString property, 395-396**
- ConnectionTimeout property, 395-396**
- consuming Web services, 60, 62-63, 274, 276, 278-284**
- Contains() method**
 - ArrayList class, 305, 314
 - Hashtable class, 337, 343
 - Queue class, 347, 351
 - SortedList class, 356, 362-363
 - SqlParameterCollection class, 453, 458-459
 - Stack class, 373, 376-377
- ContainsKey() method**
 - Hashtable class, 337, 343-344
 - SortedList class, 356, 363-364
- ContainsValue() method**
 - Hashtable class, 337, 344-345
 - SortedList class, 356, 364
- content, *see* HTML**
- ContentEncoding property, 488, 491, 508, 511**
- ContentLength property, 488, 491-492**
- ContentType property, 488, 492, 508, 511**
- contracts, 275-276**
- controls, *see* HTML controls; list controls; validation controls; Web controls**
- ControlToCompare property, 572**
- ControlToValidate property, 569**
- converting arraylists, 305, 326**
- cookies**
 - adding, 508, 516
 - domains, 481, 484
 - expiration, 481, 484-485
 - names, 481, 485
 - paths, 481, 485-486
 - secure connections, 481, 486
 - subkeys, 481, 485
 - transmission, 508, 511-512
 - values, 481, 486-487
- Cookies property**
 - HttpBrowserCapabilities class, 470, 473
 - HttpRequest class, 488, 492-493
 - HttpResponse class, 508, 511-512
- copying**
 - arraylists, 305, 314-315
 - bitarrays, 327, 331-332
 - hashtables, 337, 345
 - queues, 347, 351-352
 - sortedlists, 356, 365
 - stacks, 373, 376-378
- CopyTo() method**
 - ArrayList class, 305, 314-315
 - Hashtable class, 337, 345
 - Queue class, 347, 351-352
 - SortedList class, 356, 365
 - Stack class, 373, 377-378
- Count property**
 - ArrayList class, 305, 307
 - BitArray class, 327-328
 - Hashtable class, 337-338
 - Queue class, 347-349
 - SortedList class, 355, 357
 - SqlParameterCollection class, 453, 455
 - Stack class, 373-374
- counting**
 - arraylists, 305, 307
 - bitarrays, 327-328
 - parameters, 453, 455
- Crawler property, 470, 473-474**
- CreateCommand() method, 395, 401-402**
- CreateObject() method, 256**
- CreateParameter method, 385, 390-391**
- creating**
 - controls, 38
 - Master/Detail view, 178-184
 - user controls, 111-113
 - virtual directory, 294
 - Web services, 269-271

CssClass property, 144, 158, 524, 527
Custom validation control, 52, 201-204, 574

D

Dashed property, 581

data

binding, 106
caching, 12, 292
columns, 154
commands, 58
connection, 58, 211-212
deleting, 58, 212
DropDownList control, 101, 103-106
formatting, 162
forward-only stream, 58
inserting, 58
list controls, 130, 132-134
parameters, 212
read-only, 58
queries, 213-214
retrieving, 58, 212
returning, 385, 392-394
sets, 212
sources, 47-48, 164
stored procedures, 57, 213-214
transferring, 488, 497-498, 502
updating, 58, 212
writing, 509, 521-522

DataAdapter object, 220-223

databases

changing, 395, 400
closing, 395, 401
connections, 385, 387, 395-397
database access layer (DAL), 245
opening, 395, 402

DataBind() method, 120-121

DataBinder.Eval() method, 122

DataField property, 162

DataFormatString property, 162

DataGrid control, 41-42, 154

automatic column generation, 43
C# code, 155
column types, 161-166
editing, 43, 47
formatting, 157-161
Master/Detail view, 178-184
methods, 175
multiple-column display, 43, 45
objects, 166-174
paging, 43-44

properties, 154, 157, 171, 175

selecting, 43
sorting, 43, 45, 175-178
styles, 43
tables, 154
templates, 43
Visual Basic.NET, 156

DownList control, 40

automatic column generation, 43
columns, 148-149
configuration, 142, 144-148
editing, 43, 47
objects, 150-154
multiple-display column, 43
paging, 43
properties, 143, 148-154
selecting, 43
sorting, 43
styles, 43
templates, 43, 139

DataNavigateUrlField property, 164

DataNavigateUrlFormatString property, 164

DataReader object, 58, 214

accessing, 218-219
instantiating, 215-218
methods, 218-223

datasets, 57-58, 223-228

DataSource property, 120, 395, 397

DataTable class, 57

TextField property, 164-165

DataFormatString property, 164-165

DataView class, 120, 129-130

DateTime value, 403, 415, 448

DayHeader Style property, 537, 539

DayNameFormat property, 537, 540

DayStyle property, 537, 540

DbType property, 448-449

DCOM (Distributed Component Object Model), 59-60

debugging strings, 70

Decimal value, 403, 416-417, 448

declaring

members, 250
namespaces, 246-247

delegation, components, 244

deleting

arraylists, 305, 320-321
buffer streams, 508, 518
data, 58
data sources, 212

- headers, 508, 519
- objects
 - arraylists, 305, 313
 - hashtables, 337, 341-342, 346
 - queues, 347, 349-350, 352
 - sortedlists, 355-356, 361-362, 369-371
 - parameters, 453, 457, 462-463, 465
- deploying**
 - applications, 295
 - components, 295-296
 - Microsoft .NET framework, 10-11
 - scripting, 295
- Dequeue() method, 347, 352**
- designing ASP.NET application security, 296
- detection of configuration changes, 11
- Direction property, 448-449**
- Disabled value, 259**
- disabling ViewState, 35
- discovery files, 62, 284-285
- displaying, 569-570, 578
 - columns, 148-149, 163-166
 - data source, 164
 - links, 171
 - list items, 106-109
- Distributed Component Object Model (DCOM), 59-60**
- DLLs (dynamic link libraries), 14, 293-294**
- domains, 481, 484**
- Dotted property, 581**
- Double property, 403, 417-418, 433-434, 581**
- downlevel browsers 50**
- DropDownList control, 99-101, 103-106**
- duplicating**
 - arraylists, 305, 313-314
 - hashtables, 337, 342-343
 - queues, 347, 350-351
 - sortedlists, 355, 362
 - stacks, 373
- dynamic discovery, 285-286**
- dynamic link libraries (DLLs), 14, 293**

- E**
- e-commerce sites, 271-274
- early-bound references, 255-258**
- EcmaScriptVersion property, 470, 474**
- EconoJIT compiler, 14**

- EditCommandColumn column type, 162, 165-166**
- editing, 15, 43, 47, 157, 163**
- EditItemStyle property, 143, 157**
- EditItemTemplate property, 163**
- EditText property, 166**
- EnableClientScript property, 569-570**
- Enabled property, 524, 528**
- EnableViewState property, 524, 528**
- encapsulation, 252**
- encoding, 508, 511**
- end users, *see* users**
- End() method, 509, 519-520**
- Enqueue() method, 347, 353**
- error messages, 569, 571**
 - cascading style sheets (CSS), 52
 - Compare validation control, 196-198
 - configuration, 188-189
 - placing, 53
- errors**
 - applications, 289-290
 - formatting, 52
 - reporting, 51
- events**
 - classes, 249
 - Web controls, 37-38
 - Web forms, 26
- executing**
 - commands, 385-387
 - DLLs, 294
 - output, 509, 519-520
 - queries, 385, 391-392
 - transactions, 260-267
 - Xml Reader, 385-393-394
- expiration**
 - cookies, 481, 484-485
 - dates, 508, 512-513
 - page transmission, 508, 512
- exposing Web services, 61-62**
- expressions, 52, 121-122**
- eXtensible Markup Language, *see* XML**
- extensions, 111**

- F**
- fields**
 - classes, 247
 - columns, 162, 165
 - data sources, 162-164
 - names, 403, 425-426

- ordinal numbers, 403, 414-415, 418, 426-427
retrieving, 402, 404-405
SQL values, 403, 441-444
- FIFO structure, 347**
- file paths, 28**
- FilePath property, 488, 493**
- files**
adding, 508
ascx extension, 111
discovery, 284-285
dynamic discovery, 285-286
uploading, 488, 493-496
Web service discovery, 62
writing to output streams, 509, 522
- Files property, 488, 493-496**
- filling data sets, 212**
- FILO structures, 373**
- FirstDayOfWeek property, 537, 541**
- FixedSize() method, 305, 315-316**
- Float value, 403, 419-420, 448**
- Flush() method, 509, 520**
- Font property, 524, 528**
- Font-Bold property, 144, 158**
- Font-Italic property 144, 158**
- Font-Name property, 144, 158**
- Font-Size property, 144, 158**
- Font-Underline property, 144**
- FontInfo class, 580**
- FontSize class, 580**
- footers, 157, 162-166**
- FooterStyle property**
BoundColumn column type, 163
ButtonColumn column type, 165
DataGridView control, 157
DataList control, 143
EditCommandColumn column type, 166
HyperlinkColumn column type, 164
TemplateColumn column type, 163
- FooterTemplate template, 135, 139, 163**
- FooterText property, 162, 165-166**
- ForeColor property, 144, 158, 524, 529**
- Form property, 488, 496-497**
- form-based authentication, 298**
- formatting**
columns, 165
data, 162
DataGridView control, 157-161
errors, 52
text, 164
URLs, 164
- forms**
Button, 84-87
CheckBox, 88-90
CheckBoxList, 90-93
DropDownList, 99-101, 103-106
ImageButton, 84-87
LinkButton, 84-87
ListBox, 106-109
RadioButton, 93-96
TextBox, 96-98
variables, 488, 496-497
Windows, 284-286
- forward-only stream of data, 58**
- fragment caching, 11-12, 117, 291**
- Frames property, 470, 474-475**
- G**
- garbage collection**
manual control, 19
Microsoft .NET runtime, 16
- GC class, 20**
- generating**
columns, 154
Intermediate Language, 23-26
object references, 18-19
proxy classes, 278-281
stored procedures, 231-232
- Get() method, 327, 332-333**
- GetBoolean() method, 403, 409-410**
- GetByIndex() method, 356, 365-366**
- GetByte() method, 403, 410-411**
- GetBytes() method, 403, 411-412**
- GetChar() method, 403, 412-413**
- GetChars() method, 403, 413-414**
- GetDataTypeName() method, 403, 414-415**
- GetDateTime() method, 403, 415**
- GetDecimal() method, 403, 416-417**
- GetDouble() method, 403, 417-418**
- GetFieldType() method, 403, 418**
- GetFloat() method, 403, 419-420**
- GetGeneration() method, 20**
- GetGuid() method, 403, 420-421**
- GetInt16() method, 403, 421-422**
- GetInt32() method, 403, 423-424**
- GetInt64() method, 403, 424-425, 437-438**
- GetKey() method, 356, 366**
- GetKeyList() method, 356, 367**
- GetName() method, 403, 425-426**
- GetOrdinal() method, 218, 403, 426-427**
- GetRange() method, 305, 316**

GetSchemaTable() method, 403, 427
GetSqlBinary() method, 403, 428-429
GetSqlBit() method, 403, 429-430
GetSqlByte() method, 403, 430
GetSqlDateTime() method, 403, 431-432
GetSqlDecimal() method, 403, 432-433
GetSqlDouble() method, 403, 433-434
GetSqlGuid() method, 403, 434-435
GetSqlInt16() method, 403, 435-436
GetSqlInt32() method, 403, 436
GetSqlInt64() method, 403
GetSqlMoney() method, 403, 438-440
GetSqlSingle() method, 403, 439
GetSqlString() method, 403, 440-441
GetSqlValue() method, 403, 441-442
GetString() method, 403, 442-443
GetTotalMemory() method, 20
GetValue() method, 403, 443-444
GetValueList() method, 356, 367-368
globally unique identifiers (GUID), 6, 403, 420-421
GridLines property, 78
Groove property, 581
GroupName property, 566-567
GUID (globally unique identifiers), 6, 403, 420-421

H

hardware requirements, 296
hashtables, 120
 copying, 337, 345
 duplicating, 337, 342-343
 indexes, 337, 339-340
 keys, 337, 340, 343-344
 list controls, 125-128
 methods, 337, 341-347
 objects, 337-338, 341-342, 346
 properties, 336-341
 read-only, 337-338
 synchronization, 337-339, 346-347
 values, 337, 340-341, 343-345
HasKeys property, 481, 485
headers
 adding, 508, 516
 cache, 508, 510
 columns, 162-166
 deleting, 508, 519
 HTTP, 488, 497
 lists, 157

PICS, 509, 520-521
 properties, 488, 497
 templates, 163
HeaderStyle property, 143, 157, 163-166
HeaderTemplate property, 135, 139, 163
HeaderText property, 162-166, 578
Height property, 144, 158, 524, 529
horizontal columns, 149
HorizontalAlign property, 76, 78, 144, 158, 553-554
host addresses, 489, 505-506
HTML (Hypertext Markup Language)
 content suppression, 508, 515
 runat="server" attribute, 31, 36
 textboxes, 96
HTTP (Hypertext Transfer Protocol), 59
 character set, 508, 510
 data transfer methods, 488, 497-498
 encoding, 508, 511
 headers, 488, 497
 MIME types, 508, 511
 status, 508, 513-515
HttpBrowserCapabilities class, properties, 469-481
HttpCookie class, 481-487
HttpMethod property, 488, 497-498
HttpRequest class
 methods, 489, 507
 properties, 488-506
 requests, 507
HttpResponse class, 508-522
HyperLink class, 533-535
HyperLink control, 71-73
HyperlinkColumn type, 162, 164
Hypertext Markup Language, *see* **HTML**
Hypertext Transfer Protocol, *see* **HTTP**

I

ICollection **Datasource** property, 99
ID property, 524, 530
IIS log file, 508, 517
IL Disassembler, 15, 22-26
IList **DataSource** property, 106
Image class, 73-75, 448, 552-553
ImageAlign class, 552, 561, 563, 580-581
ImageButton class, 84-87, 561-564
ImageUrl property, 162-166, 552-553, 561, 564
implementation, components, 244

indexes

arraylists, 305, 309-310, 317, 319
bitarrays, 327, 329
collections, 453, 455-456
DataList control, 150-154
hashtables, 337, 339-340
keys, 356, 368-369
sorted lists, 355, 359
values, 356, 369

IndexOf() method, 305, 317, 453, 459, 461

IndexOfKey() method, 356, 368-369

IndexOfValue() method, 356, 369

InitialValue property, 577

input fields, 51, 228-231, 448-449

Insert() method, 305, 317-318, 453, 461-462

inserting

code, 27
data, 58
see also adding

InsertRange() method (ArrayList class), 305, 318-319

Inset property, 581

installation, ASP.NET, 294

instances, 16-17, 112-113, 215-218, 249-252

int Columns, 96

int MaxLength, 96

Int property, 448

int Rows property, 96, 106

int SelectedIndex property, 99, 106

Int value, 403, 421-425, 437-438

int X property, 84

int Y property, 84

Intermediate Language Disassembler, 15, 22-26

internal member, 252

Internet Services Manager (ISM) , 294

interoperability

ADO.NET, 55-56
programming languages, 13

IsAuthenticated property, 488, 498

IsClientConnected property, 508, 513

IsClosed property, 402, 405-406

IsDBNull() method, 403, 444-445

IsFixedSize property, 305, 307-308

IsNull property, 448, 450

IsolationLevel property, 465-466

IsReadOnly property, 305, 308, 327-328, 337-338, 355-358, 373-374

IsSecureConnection property, 488, 498

IsSynchronized property, 305, 309, 327-329, 337-339, 355, 358, 373, 375

IsValid property, 569, 571

italic attribute, 144, 158

Item property, 402, 406-407, 453, 455-456, 580

ItemStyle property, 143, 157, 163, 164, 166

ItemTemplate property, 135, 139, 164

Item[] property, 305, 309-310, 327, 329, 337, 339-340, 355, 359

J-K

Java applications, 13

JavaApplets property, 470, 475

JavaScript property, 470, 475-476

JIT compilation, 14, 22, 282

just-in-time compiling, 22, 282

KeepAlive() method, 20

keys

hashtables, 337, 340, 343-344

indexes, 356, 368-369

location, 356, 366

sorted lists, 355-356, 359-360, 363-364, 367

Keys property, 337, 340, 355, 359-360

L

Label class, 36, 69-70, 551

language

independence, 7

users, 489, 506

Large property, 580

Larger property, 580

LastIndexOf() method, 305, 319

late-bound references, 255-256

Left property, 580-581

length

arraylists, 305-306

bitarrays, 327, 330

requests, 488, 491-492

sorted lists, 355-356

Length property, 327, 330

libraries, 14-15

LinkButton class, 84-87, 564-566

links, 164, 171

list controls, 32, 122-128

bindings, 130, 132-134

browsers, 43

data sources, 47-48

DataGrid, 41-45

DataList, 40, 43, 47

Dataview object, 129-130
 performance, 42
 Repeater, 40, 43, 135, 137-139

list items, 101, 106-109

List property, 581

ListBox control, 106-109

ListBoxSelectionMode SelectionMode property, 106

listings

- Accounts object, 265-267
- ActiveXControl property, 576
- ActiveXControls, 470
- AlternateText property, 552, 562
- AOL, 471
- applications, 490, 500
- AppSettings, 288
- arraylists
 - copying, 315
 - counting, 307
 - creating, 326
 - duplicating, 313
 - indexes, 310, 317, 319
 - length, 306
 - objects, 311, 313, 318, 321
 - ranges, 311-312, 316, 318, 322, 324
 - read-only, 308, 320
 - repeating, 322
 - returning, 310
 - reversing, 323
 - searches, 312
 - sizing, 307-308, 316
 - sorting, 325
 - synchronization, 309, 325
 - trimming, 326
 - True value, 314
- authentication, 299
- AutoPostBack property, 556, 560, 567
- BackImageUrl property, 554
- beta versions, 472
- bitarrays
 - And operation, 331
 - copying, 331-332
 - counting, 327
 - indexes, 329
 - location, 333, 335
 - Not operation, 333-334
 - Or operation, 334
 - read-only, 328
 - synchronization, 329
 - values, 335-336
 - Xor operation, 336
- Boolean values, 409-410
- browsers, 472, 476, 478-479, 490
- buffers, 509, 518, 520
- Button control, 85-86
- bytes, 503
- cache, 510, 513
- CDF, 473
- CellPadding property, 538
- CellSpacing property, 539
- CheckBox control, 88-89
- CheckBoxList control, 90-92
- Checked property, 560, 567
- Clear() method, 375-376
- clients, 491, 521, 574
- Clone() method, 376
- collections, 454-455
- columns, 406-407, 556
- COM components, 256-258
- Command object, 221-223, 401
- CommandArgument property, 536, 562, 565
- CommandName property, 536, 563, 565
- CommandText property, 386
- CommandTimeout property, 386
- CommandType property, 387
- COMP proxy class, 257
- Compare validation control, 196-198
- components, 253-255
- connections, 212, 387, 498, 519
- ConnectionString property, 395
- ConnectionTimeout property, 396
- Contains() method, 377
- ControlToCompare property, 573
- ControlToValidate property, 570
- cookies, 473, 482-487, 512, 516
- CopyTo() method, 377-378
- crawler search engines, 474
- CreateParameter() method, 391
- Custom validation control, 202-204
- data, 497, 521
- DataAdapter object, 220-221
- databases, 397, 400-402
- DataGrid control, 155-156
 - applying styles, 159-161
 - objects, 166-170
 - paging, 171-174
 - sorting, 176-178
- DataGrid list control, 33-34
- DataList control, 140-142, 145-148, 151-154
- DataReader object, 215-219

datasets, 226-228
DataSource property, 397
DayHeaderStyle property, 539
DayNameFormat property, 540
DayStyle property, 540
discovery files, 284-285
Display property, 570
DisplayMode property, 578
DropDownList control, 100-105
dynamic discovery, 286
ECMA script, 474
EnableClientScript property, 571
error pages, 290, 571
ExecuteNonQuery() method, 214, 391-392
ExecuteReader() method, 392
ExecuteScalar() method, 213, 393
ExecuteXmlReader() method, 394
FieldCount property, 404
fields, 247, 414-415, 419, 425, 426
files, 493-496, 522
FirstDayOfWeek property, 541
Float values, 420
forms, 497
fragment caching, 117
frames, 475
GetBoolean() values, 409
GetByte() value, 410-412
GetChar() value, 412-414
GetDateTime() value, 415-416
GetDecimal() value, 416-417
GetDouble(), 418
GroupName property, 568
GUID value, 421
hashtables
 copying, 345
 duplicating, 342
 keys, 340, 343-344
 objects, 337-339, 341-342, 346
 read-only, 338
 removing, 346
 synchronization, 346-347
 values, 341, 344-345
headers, 497, 515-516, 519, 579
HorizontalAlign property, 554
HTML content, 515
HTTP, 278, 510-511, 514-515
HyperLink control, 71
ID property, 530
IIS log file, 517
Image control, 73-74
ImageAlign property, 552, 563
ImageButton control, 85-86
ImageUrl property, 533-534, 553, 564
InitialValue property, 577
Int value, 422-424
Intermediate Language, 23-26
IsClosed property, 405
IsValid property, 571
JavaScript, 475
Label control, 69-70
LinkButton control, 85-86
list controls, 122-134
ListBox control, 32, 106-109
login URLs, 298
Master/Detail view, 178-184
MaximumValue property, 575
MaxLength property, 556
members, 250-252
methods, 248
MIME types, 489, 492
MinimumValue property, 576
namespaces, 246
NavigateUrl property, 534
next results, 446
NextMonthText property, 541
NextPrevFormat property, 542
NextPrevStyle property, 542
Null value, 445
Operator property, 573
OtherMonthDayStyle property, 543
output, 510, 517, 520
PacketSize property, 398
pages, 512, 530
Panel control, 76-77
parameters, 449, 455, 499
 adding, 456-457, 461
 checking, 458-459
 deleting, 457-458, 462-464
 input, 228-238
 input/output, 450
 location, 460
 names, 451
 Null, 450
 output, 232-238
 precision, 451
 properties, 388-389
 scale, 452
 size, 452
 values, 453
Parent property, 531
paths, 500

Peek() method, 378
physical disks, 501
Pics() method, 521
platforms, 477
Pop() method, 379
PrevMonthText property, 543
properties, 249
Push() method, 379
queries, 390, 501
queues
 arrays, 355
 clearing, 350
 copying, 352
 counting, 348
 deleting, 350
 duplicating, 350
 objects, 352-354
 read-only, 348-349
 synchronization, 349, 354
 values, 351
RadioButton control, 94-95
Range validation control, 193-194
ReadOnly property, 557
records, 447
RecordsAffected property, 407-408
references, 256
RegularExpression validation control,
 198-201
Repeater control, 136-139
requests, 491, 492, 502, 507
RequiredField validation control,
 189-191
ResetCommandTimeout() method, 394
Rows property, 557
schema tables, 427
SelectedDate property, 544
SelectedDayStyle property, 544
SelectionMode property, 545
SelectMonthText property, 545
SelectorStyle property, 546
SelectWeekText property, 546
Send method, 276
servers, 503
ServerVersion property, 398
ShowDayHeader property, 546
ShowGridLines property, 547
ShowMessageBox property, 579
ShowNextPrevMonth property, 547
ShowSummary property, 580
ShowTitle property, 548
16-bit platforms, 480
SOAP SendMail, 277-278
sortedlists
 adding, 361
 capacity, 356-357
 copying, 365
 counting, 357
 deleting, 361-362, 370
 duplicating, 362
 indexes, 366, 368-369, 371
 keys, 360, 363-364, 366-367
 objects, 359
 read-only, 358
 removing, 370
 sizing, 372-373
 synchronization, 358-359, 372
 values, 360, 363-364, 368
sounds, 471
SqlBinary value, 428
SqlBit value, 429
SqlByte value, 430
SqlDataReader object, 408-409
SqlDateTime value, 431
SqlDecimal value, 432-433
SqlDouble value, 433-434
SqlGuid value, 434-435
SqlInt16 value, 435
SqlInt32 value, 436-437
SqlInt64 value, 437-438
SqlMoney value, 438-439
SqlSingle value, 439-440
stacks, 374-375
State property, 399
stored procedures, 224-225, 231-232,
 261
String value, 443
Synchronized() method, 380
System.Data.SqlClient namespace,
 383-385
TabIndex property, 531
tables, 79-82, 478
Target property, 534
Text property, 535, 537, 551, 558, 560,
 566, 568, 572
 TextAlign property, 561, 569
TextBox control, 97-98
TextMode property, 558
32-bit platforms, 481
TitleFormat property, 548
TitleStyle property, 549
ToArray() method, 381
TodayDayStyle property, 549

- TodaysDate property, 550
ToolTip property, 532
transactions, 258, 389
 beginning, 239
 committing, 466
 isolation level, 465
 rollbacks, 239, 467-468
 starting, 400
 support, 262-264
URLs, 502, 504
user controls, 112-117, 498, 505-506
ValidationSummary control, 206-208
values, 444, 574
VBScript, 479
ViewState management, 35
virtual paths, 499
Visible property, 532
VisibleDate property, 550
Visual Basic.NET, 436
W3C XML, 480
Web controls, 524-530
Web forms, 22-23
Web servers, 513
Web service, 61-62, 270-274, 280-284
web.config file, 288
WeekendDayStyle property, 551
Width property, 533
WorkstationId property, 399
Wrap property, 555, 559
XML documents, 477
- ListItem SelectedItem property, 99, 106**
- ListItemCollection Items property, 106**
- lists, 157**
- location**
- bitarrays, 327, 332-333, 335
 - keys, 356, 366
 - parameters, 453, 459, 461
 - physical disks, 488, 500-501
 - sortedlists, 356, 365-366
 - Web services, 63
- log files, 508, 517**
- logical operations, 327, 330-336**
- login pages, 298-299**
- M**
- machine.config file, 287**
- maintaining state, 34-35**
- maintenance, components, 244**
- MajorVersion property, 470, 476**
- managed heap, 16**
- managing ViewState, 34-35**
- manifests, 14**
- manual control, garbage collection, 19**
- MapPath() method, 489, 507**
- Mark and Compact algorithm, 16-17**
- Master/Detail view, 178-184**
- MaximumValue property, 575, 578**
- MaxLength property, 555-556**
- Medium property, 580**
- members**
- classes, 247-249, 252
 - declaring, 250
 - instance, 249-252
 - referencing, 250-251
 - shared, 249-252
 - static, 249-252
- methods**
- ArrayList class, 305, 310-326
 - BitArray class, 327, 330-336
 - Connection object, 238
 - data sources, 211
 - data transfer, 488, 497-498, 502
 - DataBind(), 120-121
 - GridView control, 175
 - DataReader object, 218-223
 - GetGeneration(), 20
 - GetTotalMemory(), 20
 - Hashtable class, 337, 341-347
 - HttpRequest class, 489, 507
 - HttpResponse class, 508, 515-522
 - ImageButton control, 84
 - in classes, 248
 - KeepAlive(), 20
 - Queue class, 347, 349-355
 - Server 256
 - SortedList class, 355, 361-373
 - SqlCommand class, 385, 390-394
 - SqlConnection class, 395, 399-402
 - SqlDataReader class, 403, 408-425, 430-438, 440-447
 - SqlParameterCollection class, 453, 456-465
 - SqlTransaction class, 465-468
 - Stack class, 373, 375-380
 - Transaction object, 241
 - user controls, 116
 - validation controls, 187-188
 - Web services, 64
- Microsoft .NET, 13**
- ASP support, 5
 - assemblies, 14

- compiling, 14
- components, 252-253
- framework, 9-11, 247-249
- garbage collection, 16
- generations, 18-19
- programming models, 5
- transactions, 260-267
- Microsoft Intermediate Language (MSIL), 6, 14**
- Microsoft Message Queuing (MSMQ), 258**
- Microsoft namespace, 245**
- Middle property, 580**
- MIME**
 - requests, 488-489, 492
 - HTTP, 508, 511
- MinimumValue property, 575, 578**
- MinorVersion property, 470, 476-477**
- mixing programming languages, 13**
- Mode attribute values, 290**
- Mode property, 171**
- Money property, 448**
- MSDomVersion property, 470, 477**
- MSIL (Microsoft Intermediate Language), 6, 14**
- MSMQ (Microsoft Message Queuing), 258**
- multiple-column display, 43, 45**

- N**
- names**
 - COM+ applications, 259
 - cookies, 481, 485
 - fields, 403, 425-426
 - fonts, 144, 158
 - parameters, 449-450
 - servers, 395, 397
 - users, 112, 489, 505-506
- namespaces, 245-247, 267**
- NavigateUrl property, 164, 533-534**
- NChar property, 448**
- NetBIOS, 395, 399**
- network interface cards (NICs), 297**
- networks, 297**
- next result, 403, 446-447**
- NextMonthText property, 537, 541**
- NextPageText property, 171**
- NextPrevFormat property, 537, 542**
- NextPrevStyle property, 537, 542**
- NextResult() method, 403, 446**
- NICs (network interface cards), 297**
- None property, 581**
- Not() method, 327, 333-334**
- NotSet property, 580-581**
- NotSupportedException, 259**
- NTText property, 448**
- NTLM authentication, 298**
- Null value, 403, 444-445, 448, 450**
- NVarChar property, 448**

- O**
- objects**
 - adding, 305, 311, 317-318, 337, 341, 347, 353, 355, 361, 373
 - ADO.NET, 56-58
 - arraylists, 305, 320-321
 - bitarrays, 327-328
 - color, 144
 - Command, 213
 - Connection, 212-213
 - DataAdapter, 220-221
 - DataGridView control, 171-178
 - DataReader, 214-219
 - deleting, 305, 313, 347, 352, 355-356, 361-362, 369-371
 - hashtables, 337-338, 341-342, 346
 - instantiating, 16-17
 - queues, 347-350, 353-354
 - references, 16-19
 - returning, 373
 - selecting, 150-154, 166-170
 - sorted lists, 355, 357
 - stacks, 373-374-376, 378
 - transactions, 385, 389
- OleDbCommand, 120, 212**
- OleDbConnection, 58, 120, 211**
- OleDbDataAdapter class, 120, 212**
- OleDbDataReader class, 120, 212**
- OleDbParameter class, 212**
- OnCheckedChanged property, 88, 93**
- OnClick() method, 84**
- OnSelectedIndexChanged Datasource property, 99, 106**
- OnTextChanged (TextBox control), 96**
- On_Click() event, 38**
- Open() method, 395, 402**
- opening databases, 395, 402**
- operating systems, 297**
- operations, 327, 330-336**
- Operator property, 572-573**
- operators, 195**
- options, 253**

Or() method, 327, 334-335
ordinal numbers, 403, 414-415, 418, 426-427
OtherMonthDay property, 537
OtherMonthDayStyle property, 542
output
 applications, 11, 291
 binary characters, 508, 517-518
 buffered, 508-510, 520
 data, 509, 521-522
 execution, 509, 519-520
 files, 509, 522
 headers, 508, 519
 parameters, 448-449
 stored procedures, 231-238

OutSet property, 581

OverLine property, 580

P

packets, sizing, 395, 397-398

Page property, 524, 530

PageButtonCount property, 171

PagerStyle property, 171

pages

- ASP.NET application, 299
- authentication, 299
- caching, 11
- directives, 21, 26
- loading, 37
- logic, 21
- transmission, 508, 512

paging controls, 43-44, 171-174

Panel class, 553-554

Panel control, 76-78

ParameterName property, 449-450

parameters

- adding, 453, 456-457, 461-462
- checking, 453, 458-459
- collections, 488, 499
- counting, 453, 455
- data sources, 212
- deleting, 453, 457, 462-463, 465
- input, 448-449
- location, 453, 459, 461
- names, 449-450
- null values, 448, 450
- output, 448-449
- precision, 449, 451
- queries, 385, 388-389

returning, 385, 390-391

scale, 449, 451

size, 449, 452

SQL, 448-449

stored procedures, 228-238

values, 449, 452

Parameters property, 385, 388-389

Params property, 488, 499

Parent property, 524, 531

passing parameters, 212

PathInfo property, 488, 500

paths

- applications, 488, 490
- cookies, 481, 485-486
- files, 28
- mapping, 489, 507
- virtual, 28, 488, 493, 499

Peek() method, 347, 353-354, 373, 378

performance

- ADO.NET, 56
- list controls, 42

PhysicalApplicationPath property, 488, 500

physical disks, 488-501, 507

Pics() method, 509, 520-521

placing error messages, 53

Platform property, 470, 477-478

platforms, 6, 470, 480-481

Pop() method, 373, 378-379

port 80, Web services, 60

Position property, 171

precision, parameters, 449, 451

precompiled classes, 282

PrevMonthText property, 537, 543

PrevPageText property, 171

private member, 252

processing transactions, 258

programming languages, 13, 14, 61, 270

programming models, 5-6

properties

- ArrayList class, 305-310
- BaseValidator class, 569-572
- BitArray class, 327-330
- BorderStyle class, 581
- BoundColumn column type, 162-163
- Button class, 84-87, 535-536
- ButtonColumn column type, 165
- Calendar class, 537-550
- CheckBox class, 559-561
- CheckBox control, 88
- Compare validation control, 195, 572-573

Custom validation control, 201, 574
 data sources, 211
 DataGrid control, 154, 157, 171, 175
 DataList control, 143, 148-154
 DataSource, 120
 DropDownList control, 99
 EditCommandColumn column type, 166
 FontInfo class, 580
 FontSize class, 580
 Hashtable class, 337-341
 HttpBrowserCapabilities, 469-481
 HttpCookie class, 481, 484-487
 HttpRequest class, 488-503, 506
 HttpResponse class, 508-515
 HyperLink class, 533-535
 HyperlinkColumn column type, 164
 Image class, 552-553
 ImageAlign class, 580-581
 ImageButton class, 84, 561-564
 in classes, 248-249
 Label class, 551
 LinkButton class, 84-87, 564-566
 list items, 109
 PagerStyle property, 171
 Panel class, 76, 553-554
 Queue class, 347-349
 RadioButton class, 93, 566-568
 Range validation control, 193, 575
 RegularExpression validation, 198-201, 576
 RequiredField validation control, 189, 577
 SortedListed class, 355-361
 SqlCommand class, 385-389
 SqlConnection class, 395-399
 SqlDataReader class, 402, 404-408
 SqlDbType class, 448
 SqlParameter class, 448-452
 SqlParameterCollection class, 453, 455-456
 SqlTransaction class, 465-466
 Stack class, 373-375
 Table control, 78, 80-83
 TableItemStyle class, 144, 158-159
 TemplateColumn column type, 163-164
 TextAlign class, 581
 TextBox class, 96, 555-558
 user controls, 114-116
 validation controls, 187

ValidationSummary class, 205, 578-579, 581
 WebControl class, 523-532
protected member, 252
protocols, deploying, 295
proxy classes, 63, 256-257
 ASP.NET application, 281
 COM proxy class, 257
 generating, 278-281
public member, 252
Push() method, 373, 379-380

Q-R

queries

canceling, 385, 390
 data sources, 213-214
 executing, 385, 391-392
 parameters, 385, 388-389
 records, 402, 407-408
 SQL, 385-386
 timeouts, 385-386, 394

queues

arrays, 347, 354-355
 copying, 347, 351-352
 duplicating, 347, 350-351
 methods, 347, 349-354
 objects, 347, 349-350-354
 properties, 347-349
 read-only, 347-349
 strings, 488, 501

synchronizing, 347, 354

RadioButton class, 93-96, 566-568

ranges, arraylists, 305, 311-312, 316, 318-319, 321-324

RangeValidator control, 51, 193-194, 575

RawUrl property, 488, 501-502

Read() method, 218-223, 403, 447

read-only data

arraylists, 305, 308, 320
 bitarrays, 327-328
 columns, 163
 data, 58
 hashtables, 337-338
 queues, 347-349
 sorted lists, 355, 357-358
 stacks, 374

reading
next results, 403, 447
XML, 58

ReadOnly property, 163, 305, 320, 555, 557

Real property (`SqldbType` class), 448

records, queries, 402, 407-408

RecordsAffected property, 402, 407-408

recordsets, 57

Redirect() method, 509, 521

references, 255-258

referencing members, 250-251

reflection, `DataBinder.Eval()` method, 122

registration, components, 15

RegularExpression validation control, 52, 198-201, 576

Remove() method
 `ArrayList` class, 305, 320-321
 `Hashtable` class, 337, 346
 `SortedListed` class, 356, 369-370
 `SqlParameterCollection` class, 453, 462

RemoveAt() method
 `ArrayList` class, 305, 321
 `SortedListed` class, 356, 370-371
 `SqlParameterCollection` class, 453, 463, 465

RemoveOnCheckedChanged property, 88, 93

RemoveOnClick() method, 84

RemoveOnSelectedIndexChanged
 Datasource property, 99, 106

RemoveOnTextChanged, 96

RemoveRange() method, 305, 321-322

removing stacks, 375-378

Repeat() method, 305, 322

RepeatColumns property, 148

RepeatDirection property, 149

Repeater list control, 40, 43, 135, 137-139

repeating arraylists, 305, 322

reporting
 applications, 289-290
 errors, 51

requests
 browsers, 488, 490
 character sets, 488, 491
 client security certificates, 488, 490-491
 data transfer methods, 488, 502
 length, 488, 491-492
 MIME, 488, 492
 saving, 507

URLs, 488, 503-504
virtual paths, 488, 499
Web forms, 21

RequestType property, 488, 502

Required value, 259

RequiredFieldValidator class, 51, 189-191, 577

RequiresNew value, 259

ResetCommandTimeout() method, 385, 394

result sets, next result, 403, 446-447

retrieving
 Byte value, 403, 410-412
 Char value, 403, 412-414
 data, 58, 212, 223-228
 `DateTime` value, 403, 415
 Decimal value, 403, 416-417
 Double value, 403, 417-418, 433-434
 fields, 402, 404-405
 Float value, 403, 419-420
 GUID value, 403, 420-421
 Int value, 403, 421-425, 437-438
 `SqlBinary` value, 403, 428-429
 `SqlBit` value, 403, 429-430
 `SqlByte` value, 403, 430
 `SqlDateTime` value, 403, 431-432
 `SqlDecimal` value, 403, 432-433
 `SqlGuid` value, 403, 434-435
 `SqlInt16` value, 403, 435-436
 `SqlInt32` value, 403, 436
 `SqlMoney` value, 403, 438-440
 `SqlSingle` value, 403, 439-441
 String value, 403, 442-443

returning
 arraylists, 305, 310
 Boolean values, 403, 409-410
 commands, 395, 401-402
 data, 385, 392-394
 Null value, 403, 444-445
 objects, 373
 parameters, 385, 390-391
 queues, 347, 353-354
 schemas, 403, 427
 stacks, 378, 380
 True value, 347, 351
 values, 385, 392-393

reusing Web services code, 8

reversing arraylists, 305, 323

rich controls, 38

Ridge property, 581

Right property, 581
RollBack() method, 465-466
 rolling back transactions, 239-241
root directory, 488, 490, 500
Rows property, 555, 557
runat="server" attribute, 22, 31, 32, 36
 running Java applications, 13, 16

S

Save() method, 465, 467-468
SaveAs() method, 489, 507
saving
 requests, 507
 transactions, 240-241, 465, 467-468
scalability, ADO.NET, 56
Scale property, 449, 451
schemas, 403, 427
scripts
 client-side, 50
 deploying, 295
searches, 305, 312
secure connections, 481, 486, 488, 498
Secure property, 481, 486
security, 61, 296-299
SelectedDate property, 537, 543
SelectedDayStyle property, 538, 544
SelectedIndex property, 150-154
SelectedItemStyle property, 143, 157
SelectedItemTemplate template, 139
selecting
 controls, 43
DownList list control, 43
 objects, 150-154, 166-170
 Repeater list control, 43
SelectionMode property, 538, 544
SelectMonthText property, 538, 545
SelectorStyle property, 538, 545
SelectWeekText property, 538, 546
separating validation controls, 51
SeparatorStyle property, 143
SeparatorTemplate template, 135, 139
Server object, 256
server-side code
 page logic, 21
runat="server" attribute, 22
 Web forms, 21-28
 validation, 202
servers, 395-398
ServerVariables property, 488, 502-503

ServerVersion property, 395, 398
serviced components, 258
Set() method, 327, 335
SetAll() method, 327, 335-336
SetByIndex() method, 356, 371
SetRange() method, 305, 323-324
shared members, 249-252
sharing assemblies, 15
ShowDayHeader property, 538, 546
ShowGridLines property, 538, 547
ShowMessageBox property, 578-579
ShowNextPrevMonth property, 538, 547
ShowSummary property, 578-579
ShowTitle property, 538, 548
Simple Object Access Protocol (SOAP), 9, 60, 62, 284
sites, Web services, 60
16-bit platforms, 470, 480
sizing
 arraylists, 305, 307-308, 315-316, 326
 fonts, 144, 158
 packets, 395, 397-398
 properties, 449, 452, 580
 sortedlists, 356, 372-373
Small property, 580
SmallDateTime property, 448
Smaller property, 580
SmallInt property, 448
SmallMoney property, 448
SOAP (Simple Object Access Protocol), 9, 60, 62, 284
Solid property, 581
Sort() method, 305, 324-325
SortData() method, 175
sorted lists
 copying, 356, 365
 duplicating, 355, 362
 keys, 356, 363-364, 367
 location, 356, 365-366
 objects, 355-356, 361-362, 369-371
 sizing, 356, 372-373
 synchronization, 356, 371-372
 True value, 356, 362-363
 values, 356, 360, 364, 367-368, 371
sorting
 arraylists, 305, 324-325
 columns, 163-166
 controls, 43, 45, 175-178
 fields, 163-166
 Repeater list control, 43
 stack items, 379-380

sounds, background, **469, 471**

** tag**, **69-70**

SQL

- field values, **403, 441-442**
- parameters, **448-449**
- queries, **385-386**
- servers, **395, 398**

SqlBinary value, **403, 428-429**

SqlBit value, **403, 429-430**

SqlByte value, **403, 430**

SqlCommand class, **212, 385-394**

SqlConnection class, **58, 211, 395-402**

SqlDataAdapter class, **212**

SqlDataReader class, **120, 403, 408**

- GetBoolean(), **409-410**
- GetByte(), **410-411**
- GetBytes(), **411-412**
- GetChar(), **412-413**
- GetChars(), **413-414**
- GetDataTypeName(), **414-415**
- GetDateTime(), **415**
- GetDecimal(), **416-417**
- GetDouble(), **417-418**
- GetFieldType(), **418**
- GetFloat(), **419-420**
- GetGuid(), **420-421**
- GetInt16(), **421-422**
- GetInt32(), **423-424**
- GetInt64(), **424-425, 437-438**
- GetName(), **425-426**
- GetOrdinal(), **426-427**
- GetSchemaTable(), **427**
- GetSqlBinary(), **428-429**
- GetSqlBit(), **429-430**
- GetSqlByte(), **430**
- GetSqlDateTime(), **431-432**
- GetSqlDecimal(), **432-433**
- GetSqlDouble(), **433-434**
- GetSqlGuid(), **434-435**
- GetSqlInt16(), **435-436**
- GetSqlInt32(), **436**
- GetSqlInt64(), **403**
- GetSqlMoney(), **438-440**
- GetSqlSingle(), **439**
- GetSqlString(), **440-441**
- GetSqlValue(), **441-442**
- GetString(), **442-443**
- GetValue(), **443-444**
- IsDBNull(), **444-445**

SqlDataReader class, **212, 403, 408**

SqlDatetime value, **403, 431-432**

SqlDbType class, **448**

SqlDecimal value, **403, 432-433**

SqlGuid value, **403, 434-435**

SqlInt16 value, **403, 435-436**

SqlInt32 value, **403, 436**

SqlMoney value, **403, 438-440**

SqlParameter class, **212, 448-452**

SqlParameterCollection class, **453, 456-465**

SqlSingle value, **403, 439**

SqlString value, **403, 440-441**

SqlTransaction class, **465-468**

stacks

- arrays, **380**
- copying, **373, 376-378**
- duplicating, **373**
- FILO structures, **373**
- items, **378-380**
- objects, **373-376, 378**
- read-only, **374**
- synchronization, **373, 375, 380**

staging components, **295-296**

starting transactions, **395, 399-400**

state

- connections, **395, 398-399**
- maintaining, **34-35**

static members, **249-252**

Status property, **508, 513-515**

StatusCode property, **508**

StatusDescription property, **508, 514-515**

stopping output execution, **509, 519-520**

stored procedures

- AccountsDebit, **261**
- data sources, **213-214**
- datasets, **223-228**
- generating, **231-232**
- parameters, **228-238**

storing

- applications, **288**
- data, **57**
- object references, **16-17**

Strikeout property, **580**

String properties

- BackImageUrl, **76, 78**
- BubbleArgument, **84**
- BubbleCommand, **84**

- ClientValidationFunction, 201
 - ControlToCompare, 195
 - ControlToValidate, 187
 - DataFieldText, 99, 106
 - DataFieldValue, 99, 106
 - ErrorMessage, 187
 - GroupName, 93
 - HeaderText, 205
 - ImageUrl, 71
 - InitialValue, 189
 - MaximumControl, 193
 - MaximumValue, 193
 - MinimumControl, 193
 - MinimumValue, 193
 - NavigateUrl, 71
 - Target, 71
 - Text, 69, 71, 84, 88, 93, 96, 187
 - ValidationExpression, 198-201
 - value, 403, 442-443
 - ValueToCompare, 195
 - strings, debugging, 70**
 - Strong Name Utility, 264**
 - styles**
 - columns, 165-166
 - controls, 43
 - lists, 157
 - subkeys, cookies, 481, 485**
 - support, transactions, 262-264**
 - Supported value, 259**
 - SuppressContent property, 508, 515**
 - synchronization**
 - arraylists, 305, 309, 325
 - bitarrays, 327-329
 - hashtables, 337-339, 346-347
 - queues, 347, 354
 - sorted lists, 355-358, 371-372
 - stacks, 373, 375, 380
 - Synchronized() method, 325, 337, 346-347, 354, 356, 371-373, 380**
 - System namespace, 20, 245**
 - System.Collections namespace, 303-307, 309-326**
 - System.Data.OleDb, 120**
 - System.Data.SqlClient namespace, 120, 211-212, 383-385**
 - System.EnterpriseService namespace, 267**
 - System.SqlClient namespace, 385**
 - System.Web namespace**
 - HttpBrowserCapabilities class, 469-481
 - HttpCookie class, 481-487
 - HttpRequest class, 487-507
 - HttpResponse class, 508-522
 - Systems.Collection namespace**
 - BitArray class, 326, 328-336
 - Queue class, 347-355
 - SortedList class, 355-373
 - Stack class, 373-380
- T**
- TabIndex property, 524, 531**
 - Table control, 78, 80-83**
 - TableItemStyle class, 144-145, 158-159**
 - TableRowCollection Rows property, 78**
 - tables**
 - cells, 159
 - GridView control, 154
 - property, 470, 478
 - Target property, 164, 533-534**
 - TemplateColumn column type, 162-164**
 - templates, 43, 135, 139, 163**
 - testing Web services, 276-278**
 - text**
 - columns, 165-166
 - editing, 166
 - formatting, 164
 - SQL queries, 385-386
 - updating, 166
 - wrapping, 145, 159
 - Text property**
 - BaseValidator class, 569, 572
 - Button class, 165, 535-536
 - CheckBox class, 559-560
 - HyperLink class, 164, 535
 - Label class, 551
 - LinkButton class, 564, 566
 - RadioButton class, 566, 568
 - SqlDbType class, 448
 - TextBox class, 555, 558
 - .TextAlign property, 88, 93, 559, 561, 566, 568, 581**
 - TextBox class, 96-98, 555-558**
 - TextBoxMode TextMode, 96**
 - TextMode property, 555, 558**
 - TextTop property, 581**
 - 32-bit platforms, 470, 481**
 - timeouts, 385-386, 394-396**
 - Timestamp property, 448**
 - TinyInt property, 448**
 - TitleFormat property, 538, 548**
 - TitleStyle property, 538, 548**

ToArray() method, **305, 326, 347, 354-355, 373, 380**

TodayDayStyle property, **538, 549**

TodaysDate property, **538, 549**

ToolTip property, **524, 532**

Top property, **581**

TotalBytes property, **488, 503**

transactions

- aborting, 259
- beginning, 238
- classes, 259
- committing, 241, 259, 465-466
- data, 238
- executing, 260-267
- isolation level, 465-466
- objects, 385, 389
- processing, 258
- rolling back, 239-241, 465-466
- saving, 240-241, 465, 467-468
- starting, 395, 399-400
- support, 262-264

transmission

- cookies, 508, 511-512
- page expiration, 508, 512

TrimTosize() method, **305, 326, 356, 372-373**

troubleshooting, **22, 32**

True value, **305, 308, 314, 337, 343, 347, 351, 356, 362-363**

Type Library Importer utility, **256**

Type property, **470, 478-479**

U

underline attribute, **144, 158**

Underline property, **580**

Unicode data, **448**

UniqueID property, **448, 524**

Unit CellPadding property, **78**

Unit CellSpacing property, **78**

updating

- data, 58, 212
- text, 166

uplevel browsers, **50**

uploading files, **488, 493-496**

UrlReferrer property, **489, 504**

URLs (Uniform Resource Locators)

- clients, 509, 521
- column headers, 162-166
- data source, 164

discovery files, 285

formatting, 164

login, 298

physical disks, 488, 501

property, 488, 503-504

requests, 488, 503-504

unmodified, 488, 501-502

user controls

- creating, 111-112
- fragment caching, 117
- instances, 112-113
- methods, 116
- naming, 112
- properties, 114-116
- Web forms, 28

users

- agents, 489, 504
- authentication, 299, 488, 498
- host names, 489, 505-506
- languages, 489, 506
- validation controls, 49-50
- Web services, 61

V

Validate() method, **187-188**

validation controls, **8, 32, 49-50**

- client-side scripts, 50
- Compare, 195-198
- CompareValidator, 52
- Custom, 201-204
- CustomValidator, 52
- downlevel browsers, 50
- error messages, 51, 53
- methods, 187-188
- properties, 187
- Range, 193-194
- RangeValidator, 51
- RegularExpression, 198-200
- RegularExpressionValidator, 52
- RequiredField, 189-191
- RequiredFieldValidator, 51
- separating, 51
- uplevel browsers, 50
- Web forms, 188

ValidationCompareOperator Operator property, **195**

ValidationDataType Type property, **193, 195**

ValidationDisplay Display property, **187**

ValidationExpression property, **576**

ValidationSummary class, 205, 578-579
ValidationSummaryDisplayMode class, 205, 581
Value property, 449, 452, 481, 486-487
values

- arraylists, 305, 314
- bitarrays, 327, 335-336
- Boolean, 403, 409-410
- Byte, 403, 410-412
- Char, 403, 412-414
- comparing, 52
- cookies, 481, 486-487
- DateTime, 403, 415
- Decimal, 403, 416-417
- Double, 403, 417-418, 433-434
- fields, 403, 443-444
- Float, 403, 419-420
- GUID, 403, 420-421
- hashtables, 337, 340-341, 343-345
- indexes, 356, 369
- Int, 403, 421-425, 437-438
- Null, 403, 444-445
- parameters, 449, 452
- ranges, 51
- returning, 385, 392-393
- sorted lists, 355, 360-361
- sortedlists, 356, 360, 364, 367-368, 371
- SQL fields, 403, 441-442
- SqlBinary, 403, 428-429
- SqlBit, 403, 429-430
- SqlByte, 403, 430
- SqlDateTime, 403, 431-432
- SqlDecimal, 403, 432-433
- SqlGuid, 403, 434-435
- SqlInt16, 403, 435-436
- SqlInt32, 403, 436
- SqlMoney, 403, 438-440
- SqlSingle, 403, 439
- SqlString, 403, 440-441
- String, 403, 442-443
- TransactionOption, 259
- True, 305, 308, 314, 337, 343, 347, 351

Values property, 335, 337, 340-341, 360-361, 481, 487

ValueToCompare property, 572-573

VarBinary property, 448

VarChar property, 448

variables, 28, 488, 496-497, 502-503

Variant property, 448

VBScript property, 470, 479

Version property, 470, 479

versions, 5, 37, 395, 398

vertical columns, 149

VerticalAlign property, 144, 158

ViewState, 8, 34-35

virtual directory, 294

virtual paths, 28, 488, 493, 499

Visible property, 163-166, 524, 532

VisibleDate property, 538, 550

Visual Basic, 256-258, 336

Visual Basic.NET, 274

- ActiveXControls, 470, 576
- AlternateText property, 552, 562
- AOL, 471
- applications, 490, 500
- arraylists

 - copying, 315
 - counting, 307
 - creating, 326
 - duplicating, 314
 - indexes, 310, 317, 319
 - length, 306
 - objects, 311, 313, 318, 321
 - ranges, 312, 316, 318, 322, 324
 - read-only, 308, 320
 - repeating, 322
 - returning, 310
 - reversing, 323
 - searches, 312
 - sizing, 308, 316
 - sorting, 325
 - synchronization, 309, 325
 - trimming, 326
 - True value, 314

- AutoPostBack property, 556, 560, 567
- BackImageUrl property, 554
- beta versions, 472
- bitarrays, 327, 329, 331-336
- Boolean values, 409-410
- browsers, 472, 476, 478-479, 490
- buffer streams, 518
- buffers, 509, 520
- Button control, 86
- bytes, 503
- cache, 510, 513
- CDF, 473
- CellPadding property, 538
- CellSpacing property, 539
- CheckBox control, 89
- CheckBoxList control, 92

Checked property, 560, 567
Clear() method, 376
clients, 491, 521
ClientValidationFunction property, 574
Clone() method, 376
collections, 454-455
columns, 407, 556
Command object, 222-223, 402
CommandArgument property, 536, 562, 565
CommandName property, 536, 563, 565
CommandText property, 386
CommandTimeout property, 386
 CommandType property, 387
components, 254-255
connections, 212, 387, 396, 498, 519
Contains() method, 377
ControlToCompare property, 573
ControlToValidate property, 570
cookies, 473, 483-487, 512, 516
CopyTo() method, 378
crawler search engines, 474
CreateParameter() method, 391
data, 497, 522
DataAdapter object, 220-221
Database property, 397
databases, 400-402
DataGrid control, 156
 applying styles, 160-161
 objects, 168-170
 paging, 173-174
 sorting, 177-178
DataList control, 141-142, 146-148, 152-154
DataReader object, 216-217, 219
datasets, 227-228
DataSource property, 397
DayHeaderStyle property, 539
DayNameFormat property, 540
DayStyle property, 540
Display property, 570
DisplayMode property, 578
DropDownList control, 100-101, 104-105
ECMA script, 474
EnableClientScript property, 571
ErrorMessage property, 571
ExecuteNonQuery() method, 214, 392
ExecuteReader() method, 392
ExecuteScalar() method, 213, 393
ExecuteXmlReader() method, 394
FieldCount property, 404
fields, 247, 415, 419, 425, 426
files, 493, 495-496
FirstDayOfWeek property, 541
Float value, 420
forms, 497
frames, 475
GetByte() values, 410
GetBytes() value, 412
GetChar() value, 413
GetChars() value, 414
GetDateTime() value, 416
GetDecimal() value, 417
GetDouble(), 418
GroupName property, 568
GUID value, 421
hashtables
 duplicating, 342
 keys, 340, 343-344
 objects, 338-339, 341-342
 read-only, 338
 removing, 346
 synchronization, 347
 values, 341, 345
headers, 497, 515-516, 519, 579
HorizontalAlign property, 554
HTML content, 515
HTTP, 510, 511, 514-515
ID property, 530
IIS log file, 517
Image control, 74
ImageAlign property, 552, 563
ImageButton control, 86
ImageUrl property, 534, 553, 564
InitialValue property, 577
Int value, 423-424
IsClosed property, 405
IsValid property, 571
Java applets, 475
JavaScript, 475
Label control, 70
LinkButton control, 86
list controls, 124-125, 127-128, 130, 133-134
ListBox control, 108-109
Master/Detail view, 181-184
MaximumValue property, 575
MaxLength property, 556
members, 250-252

methods, 248
 MIME types, 489, 492
 MinimumValue property, 576
 namespaces, 246
 NavigateUrl property, 534
 next results, 446
 NextMonthText property, 541
 NextPrevFormat property, 542
 NextPrevStyle property, 542
 Null value, 445
 Operator property, 573
 OtherMonthDayStyle property, 543
 output, 510, 517, 520
 PacketSize property, 398
 Page property, 530
 pages, 512
 Panel control, 77
 parameters, 388-389, 455, 499
 adding, 457, 461
 checking, 459
 deleting, 458, 463-464
 input, 230-231, 235-238
 input/output, 450
 location, 460
 names, 451
 null, 450
 output, 235-238
 precision, 451
 size, 452
 values, 453
 paths, 500
 Peek() method, 378
 physical disks, 501
 Pics() method, 521
 platforms, 477
 Pop() method, 379
 PrevMonthText property, 543
 properties, 249
 Push() method, 379
 queries, 390, 501
 queues
 arrays, 355
 clearing, 350
 copying, 352
 counting, 348
 duplicating, 350
 objects, 352-354
 read-only, 349
 synchronization, 349, 354
 values, 351
 RadioButton control, 95
 ReadOnly property, 557
 records, 408, 447
 references, 256
 Repeater control, 138-139
 requests, 491, 492, 502, 507
 ResetCommandTimeout() method, 394
 Rows property, 557
 schema tables, 427
 SelectedDate property, 544
 SelectedDayStyle property, 544
 SelectionMode property, 545
 SelectMonthText property, 545
 SelectorStyle property, 546
 SelectWeekText property, 546
 servers, 398, 503
 ShowDayHeader property, 546
 ShowGridLines property, 547
 ShowMessageBox property, 579
 ShowNextPrevMonth property, 547
 ShowSummary property, 580
 ShowTitle property, 548
 16-bit platforms, 480
 sortedlists
 adding, 361
 capacity, 357
 copying, 365
 counting, 357
 deleting, 362, 370
 duplicating, 362
 indexes, 366, 368-369, 371
 keys, 360, 367
 objects, 359
 read-only, 358
 sizing, 373
 synchronization, 359, 372
 values, 360, 363-364, 368
 sounds, 471
 SqlDbType value, 428
 SqlBit value, 429
 SqlByte value, 430
 SqlDataReader object, 409
 SqlDbType value, 431
 SqlDbType value, 433
 SqlDbType value, 434
 SqlDbType value, 435
 SqlDbType value, 437
 SqlDbType value, 438
 SqlDbType value, 439
 SqlDbType value, 440

stacks, 374-375
State property, 399
stored procedures, 224-225
String value, 443
Synchronized() method, 380
System.Data.SqlClient namespace,
 384-385
TabIndex property, 531
Table control, 81-82
tables, 478
Target property, 534
Text property, 535, 537, 551, 558, 560,
 566, 568, 572
TextAlign property, 561, 569
TextBox control, 98
TextMode property, 558
32-bit platforms, 481
TitleFormat property, 548
TitleStyle property, 549
ToArray method, 381
TodayDayStyle property, 549
TodaysDate property, 550
ToolTip property, 532
transactions, 239, 241, 258, 259, 389,
 400, 465-468
URLs, 504, 502
users, 114-115, 498, 505-506
values, 444, 574
VBScript, 479
virtual paths, 499
Visible property, 532
VisibleDate property, 550
W3C XML, 480
Web controls, 524-530
Web servers, 513
Web services, 62, 270, 272
WeekendDayStyle property, 551
Width property, 533
WorkstationId property, 399
Wrap property, 555, 559
XML documents, 477

Visual Studio 6.0, 284**W****W3CDomVersion property, 470, 480****Web Control class properties**

AccessKey, 523-524
Attributes, 523, 525
BackColor, 523, 525

BorderColor, 523, 525
BorderStyle, 523, 526
BorderWidth, 523, 526
ClientID, 523, 527
CssClass, 524, 527
Enabled, 524, 528
EnableViewState, 524, 528
Font, 524, 528
ForeColor, 524, 529
Height, 524, 529
ID, 524, 530
Page, 524, 530
Parent, 524, 531
TabIndex, 524, 531
ToolTip, 524, 532
UniqueID, 524
Visible, 524, 532
Width, 524, 532

Web controls, 31, 36

adding, 36
base class, 37
browsers, 37
Button, 84-87
CheckBox, 88-90
CheckBoxList, 90-93
DropDownList, 99-101, 103-106
event handling, 37
HyperLink, 71-73
Image, 73-75
ImageButton, 84-87
Label, 36, 69-70
LinkButton, 84-87
ListBox, 106-109
On_Click() event, 38
Page_Load() event, 37
Panel, 76-78
RadioButton, 93-96
Table, 78, 80-83
TextBox, 96-98
versions, 37

Web forms

client-side scripts, 21
code, 21
compiling, 22-26
events, 26
Intermediate Language, 22
page directives, 21, 26
requests, 21
see also ASP.NET pages, 7
server-side, 21-28

- System.EnterpriseServices namespace, 267
user controls, 28
validation controls, 188
Visual Studio, 28
- Web server connections, 508, 513**
- Web Service Description Language (WSDL), 60, 275-276, 279-280**
- Web services, 8**
- ASP, 284
 - calculations, 60
 - calling, 64
 - classes, 61, 270
 - code, 8
 - consuming, 60, 62-63, 274, 276, 282-284
 - creating, 269-271
 - DCOM, 59-60
 - discovery files, 62, 284-285
 - dynamic discovery, 285-286
 - e-commerce sites, 271-274
 - end users, 61
 - exposing, 61-62
 - port 80, 60
 - precompiled classes, 282
 - programming languages, 61, 270
 - proprietary code, 60
 - proxy classes, 63, 278-281
 - security, 61
 - sites, 60
 - SOAP, 9, 60, 62
 - testing, 276-278
 - Windows forms, 284-286
 - WSDL, 275-276, 279-280
- web sites, CERT Coordination Center, 297
- web.config files, 11, 267, 288**
- WebServiceUtil utility, 63**
- WeekendDayStyle property, 550**
- Width property, 144, 145, 158, 159, 524, 532**
- Win16 property, 470, 480**
- Win32 property, 481**
- Window NT LAN Manager (NTLM), 298**
- Windows, 284-286**
- Windows Distributed Internet Applications (DNA), 244-245**
- WorkstationId property, 395, 399**
- workstations, NetBIOS identifiers, 395, 399**
- Wrap property, 145, 159, 553-555, 558**
- writing**
- binary characters, 508, 517-518
 - data to output streams, 509, 521-522
 - files to output streams, 509, 522
 - XML, 58
- WSDL (Web Service Description Language), 60, 275-276, 279-280**
- X-Z**
- XLarge property, 580**
- XML, 58, 130, 132-134, 284**
- XMLDataReader object, 385, 393-394**
- Xor() method, 327, 336**
- XSmall property, 580**
- XXLarge property, 580**
- XXSmall property, 580**