# Programming Applications for Microsoft® Office Outlook® 2007

*Randy Byrne; Ryan Gregg*

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/9179.aspx

9780735622494
Publication Date: February 2007

Microsoft Press

# Table of Contents

**Foreword** . . . . . . . . . . xxi
**Acknowledgments** . . . . . . . . . . xxv
**Introduction** . . . . . . . . . . xxvii
Why We Wrote This Book . . . . . . . . . . xxvii
Who This Book Is For . . . . . . . . . . xxviii
How This Book Is Organized . . . . . . . . . . xxviii
Part I: Introducing Microsoft Office Outlook 2007 . . . . . . . . . . xxviii
Part II: Quick Guide to Building Solutions . . . . . . . . . . xxix
Part III: Working with Outlook Data . . . . . . . . . . xxix
Part IV: Providing a User Interface for Your Solution . . . . . . . . . . xxix
Part V: Advanced Topics . . . . . . . . . . xxx
Sample Code on the Web . . . . . . . . . . xxx
Code Snippets . . . . . . . . . . xxxiii
Building the Sample Add-Ins . . . . . . . . . . xxxiii
System Requirements . . . . . . . . . . xxxv
Support for This Book . . . . . . . . . . xxxv

Part I **Introducing Microsoft Office Outlook 2007**

1 **What's New in Microsoft Office Outlook 2007** . . . . . . . . . . 3
Form Regions . . . . . . . . . . 4
Security . . . . . . . . . . 6
*Table* Object . . . . . . . . . . 7
Improved Search . . . . . . . . . . 8
Enhanced Events . . . . . . . . . . 9
*AddressEntry* Enhancements . . . . . . . . . . 11
*SelectNamesDialog* Object . . . . . . . . . . 11
*ExchangeUser* and *ExchangeDistributionList* Objects . . . . . . . . . . 12

**What do you think of this book? We want to hear from you!**

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

**www.microsoft.com/learning/booksurvey/**

Sharing Objects . . . . . . . . . . 12
Rules Objects . . . . . . . . . . 13
*PropertyAccessor* Object . . . . . . . . . . 14
*PropertyAccessor* Sample Code . . . . . . . . . . 14
Developer Reference . . . . . . . . . . 16
Summary . . . . . . . . . . 17

**2 Outlook as a Platform . . . . . . . . . . 19**

Why Integrate with Outlook? . . . . . . . . . . 19
Different Types of Outlook Integration . . . . . . . . . . 21
Data Integration . . . . . . . . . . 22
Functional Integration . . . . . . . . . . 24
Integration Guidelines . . . . . . . . . . 26
Data Integration . . . . . . . . . . 26
Business Logic . . . . . . . . . . 29
User Interface Integration and Data Presentation . . . . . . . . . . 30
InfoPath Forms . . . . . . . . . . 38
APIs . . . . . . . . . . 40
Architecture . . . . . . . . . . 40
Outlook Object Model . . . . . . . . . . 41
Form Regions . . . . . . . . . . 43
MAPI as a Platform Component . . . . . . . . . . 45
Outlook 2007 Integration API Reference . . . . . . . . . . 51
Simple MAPI . . . . . . . . . . 51
Deemphasized and Phased-Out Components . . . . . . . . . . 52
Development Tools . . . . . . . . . . 53
Visual Basic for Applications . . . . . . . . . . 53
Visual Studio Tools for Office . . . . . . . . . . 54
Managed Versus Native Code . . . . . . . . . . 55
Add-In Model . . . . . . . . . . 56
Summary . . . . . . . . . . 57

**Part II Quick Guide to Building Solutions**

**3 Writing Your First Outlook Add-in Using Visual Basic .NET . . . . . . . . . . 61**

Introducing the Instant Search Add-In . . . . . . . . . . 61
Install the Outlook Add-in Templates . . . . . . . . . . 62

Creating the Instant Search Add-In . . . . . 63
Writing Code . . . . . 65
The *InitializeAddin* Method . . . . . 66
Turn *Option Strict* On . . . . . 67
Adding Instance Variables . . . . . 67
Hooking Up Events in Visual Basic . . . . . 68
*ItemContextMenuDisplay* Event . . . . . 68
*ContextMenuClose* Event . . . . . 70
The *DisplayInstantSearchExplorer* Method . . . . . 71
Writing Code for Submenu *Click* Events . . . . . 72
Building the Add-in Project . . . . . 73
Creating a Shim Project . . . . . 74
Creating a Setup Project . . . . . 78
Building the Setup Project . . . . . 81
Installing the Instant Search Add-In . . . . . 81
Testing the Instant Search Add-in Solution . . . . . 82
What to Expect . . . . . 82
Troubleshooting . . . . . 82
Debug Mode . . . . . 83
Debugging Code . . . . . 84
Summary . . . . . 85

**4 Writing Your First Outlook Add-in Using C# . . . . . 87**

Introducing the Instant Search Add-In . . . . . 87
Install the Outlook Add-in Templates . . . . . 88
Creating the Instant Search Add-In . . . . . 89
Writing Code . . . . . 91
*InitializeAddin* Method . . . . . 92
Adding Instance Variables . . . . . 93
Hooking Up Events in Visual C# . . . . . 94
*ItemContextMenuDisplay* Event . . . . . 94
*ContextMenuClose* Event . . . . . 97
Cleaning Up Event Handlers . . . . . 98
*DisplayInstantSearchExplorer* Method . . . . . 98
Writing Code for Submenu *Click* Events . . . . . 99
Building the Add-in Project . . . . . 101
Creating a Shim Project . . . . . 101

Creating a Setup Project . . . 105
Building the Setup Project . . . 108
Installing the Instant Search Add-In . . . 108
Testing the Instant Search Add-in Solution . . . 109
What to Expect. . . . 109
Troubleshooting . . . 109
Debug Mode . . . 110
Debugging Code . . . 111
Summary . . . 112

Part III **Working with Outlook Data**

**5 Built-in Item Types . . . 115**
Introduction to Built-in and Custom Item Types . . . 115
Understanding *MessageClass* . . . 117
Built-in vs. Custom Types . . . 118
Creating an Item . . . 118
*MailItem*, *PostItem*, and *SharingItem* Objects . . . 122
Appropriate Uses of *MailItem* and *PostItem* . . . 123
Compose *MailItem* . . . 123
Read *MailItem* . . . 132
Adding an Electronic Business Card . . . 136
Create a To-Do Item . . . 137
*PostItem* Object . . . 139
Creating a *PostItem* . . . 140
Responding to a *PostItem* . . . 140
*AppointmentItem* Object . . . 140
Appropriate Uses of *AppointmentItem*. . . . 141
One-Time Appointments . . . 141
All-Day Events . . . 143
Appointment Attendees . . . 144
Recurring Appointments . . . 146
*MeetingItem* Object . . . 154
*ContactItem* Object . . . 158
Appropriate Uses of *ContactItem* . . . 158
Working with Contact Properties . . . 158
Electronic Business Cards . . . 160

*TaskItem* Object . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .162
Appropriate Uses of *TaskItem*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .162
Creating a Recurring Task . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .162
Delegating a Task . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .163
*TaskRequestItem* Object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .163
Working with Task Requests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .164
Other Item Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .166
*DistListItem* Object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .166
*JournalItem* Object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .167
*NoteItem* Object . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .167
*StorageItem* Object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .168
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .170

**6 Accessing Outlook Data. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 171**

An Overview of Outlook Data Storage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .171
Exchange Server . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .171
Personal Folder Files (.pst) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .173
Custom Store Providers. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .173
*Accounts* Collection and *Account* Object . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .173
*Stores* Collection and *Store* Object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .174
*Stores* Collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .174
Adding or Removing a *Store* Programmatically . . . . . . . . . . . . . . . . . . . . . . . .175
Working with the *Store* Object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .176
*Folders* Collection and *Folder* Objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .178
An Overview of Folder Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .178
*Folders* Collection. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .180
*Folder* Object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .182
Working with the *Folder* Object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .183
*Folder* Properties and Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .187
Folder Permissions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .191
Assigning Folder Permissions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .192
Assigning Roles. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .193
Using the *SharingItem* Object to Assign Folder Permissions. . . . . . . . . . . . . .194
Accessing Items in a Folder. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .194
Performance Considerations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .194
*OutlookItem* Helper Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .195
*Items* Collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .196

*Table* Object . . . 201
Summary . . . 214

7 **Address Books and Recipients . . . 215**
An Overview of Outlook Address Books . . . 215
Exchange Global Address List . . . 215
Exchange Containers . . . 216
Offline Address Book . . . 216
Outlook Address Book . . . 217
Other Address Book Providers . . . 217
The *Recipients* Collection and *Recipient* Objects . . . 218
Outlook Object Model Guard Considerations . . . 218
The *CreateRecipient* Method . . . 218
Working with the *Recipients* Collection Object . . . 220
Obtaining the SMTP Address of a Recipient . . . 223
The *AddressLists* Collection and *AddressList* Objects . . . 224
Enumerating *AddressList* Objects . . . 224
The *AddressListType* Property . . . 224
Determining Resolution Order of Address Lists . . . 225
Finding a Specific *AddressList* Object . . . 225
Determining the Contacts Folder for a Contacts Address Book . . . 226
The *AddressEntries* Collection and *AddressEntry* Object . . . 227
The *AddressEntryUserType* Property . . . 228
Finding a Specific *AddressEntry* Object . . . 229
The *GetAddressEntryFromID* Method . . . 229
Displaying *AddressEntry* Details . . . 231
Getting Availability Information for a User . . . 232
The *ExchangeUser* Object . . . 234
Working with *ExchangeUser* Properties . . . 234
Obtaining an *ExchangeUser* Object from an *AddressEntry* Object . . . 235
The *GetExchangeUserManager* Method . . . 236
The *GetDirectReports* Method . . . 236
The *GetMemberOfList* Method . . . 237
Obtaining Proxy Addresses for an *ExchangeUser* Object . . . 238
The *ExchangeDistributionList* Object . . . 238
The *GetExchangeDistributionListMembers* Method . . . 239
The *GetMemberOfList* Method . . . 240

The *GetOwners* Method . . . 240
The *SelectNamesDialog* Object . . . 240
Using the *SetDefaultDisplayMode* Method . . . 241
Dialog Caption and Recipient Selectors . . . 242
Setting the *InitialAddressList* Property . . . 243
Displaying the Select Names Dialog Box . . . 245
Using *SelectNamesDialog.Recipients* . . . 245
Summary . . . 246

**8 Responding to Events . . . 247**

Writing Event Handlers in Managed Code . . . 247
Hooking Up Events in Visual Basic .NET . . . 249
Hooking Up Events in C# . . . 251
Outlook 2007 Events . . . 254
*Application* Object Events . . . 254
*Explorers* Collection Event . . . 259
*Explorer* Object Events . . . 262
*Folders* Collection Events . . . 264
*Folder* Object Events . . . 264
*FormRegion* Object . . . 265
*Inspectors* Collection Event . . . 265
*Inspector* Object Events . . . 268
*Items* Collection Events . . . 269
Item-Level Events . . . 270
*Namespace* Object Events . . . 274
*NavigationGroups* Collection Events . . . 275
*NavigationPane* Object Event . . . 275
*OutlookBarPane* Object Events . . . 275
*OutlookBarGroup* Object Events . . . 276
*OutlookBarShortcut* Object Events . . . 276
*Stores* Collection Events . . . 277
*SyncObject* Object Events . . . 278
*Reminders* Collection Events . . . 278
*Views* Collection Events . . . 279
Summary . . . 280

**9 Sharing Information with Other Users . . . 281**

Outlook and Shared Data . . . 281

Sharing in iCalendar Format . . . . . . . . . . 281
Sharing a Calendar Through E-Mail . . . . . . . . . . 282
Saving a Calendar to Disk. . . . . . . . . . 283
Saving an Appointment to Disk . . . . . . . . . . 284
Opening an iCalendar File . . . . . . . . . . 285
Subscribing to Shared Folders . . . . . . . . . . 286
RSS Feeds . . . . . . . . . . 286
SharePoint Folders. . . . . . . . . . 287
Internet Calendars . . . . . . . . . . 289
Using the *SharingItem* Object . . . . . . . . . . 290
*SharingItem* Types . . . . . . . . . . 291
Sharing a Folder with a Sharing Invitation . . . . . . . . . . 291
Requesting Folder Access with a Sharing Request. . . . . . . . . . 292
Processing a Sharing Item . . . . . . . . . . 293
Summary. . . . . . . . . . 295

**10 Organizing Outlook Data. . . . . . . . . . 297**

How Outlook 2007 Helps to Organize Information. . . . . . . . . . 297
The *Categories* Collection and *Category* Objects . . . . . . . . . . 297
Creating a Category . . . . . . . . . . 299
Assigning One or More Categories to an Item. . . . . . . . . . 299
Displaying the Categories Dialog Box . . . . . . . . . . 300
Task Flagging . . . . . . . . . . 301
Controlling Visibility of the To-Do Bar . . . . . . . . . . 301
Creating To-Do Items That Appear in the To-Do Bar. . . . . . . . . . 302
The *Rules* Collection and *Rule* Objects. . . . . . . . . . 303
Overview of Rules Programming . . . . . . . . . . 303
*Rules* Collection . . . . . . . . . . 306
The *Rule* Object . . . . . . . . . . 310
The *RuleActions* Collection. . . . . . . . . . 312
The *RuleConditions* Collection. . . . . . . . . . 314
Get or Set Action or Condition Properties with an Array . . . . . . . . . . 317
Rules Sample Add-In. . . . . . . . . . 318
Search Folders . . . . . . . . . . 319
When to Use a Search Folder. . . . . . . . . . 319
Enumerating Search Folders . . . . . . . . . . 320
Creating a Search Folder Programmatically . . . . . . . . . . 321

Outlook Views. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .325
Objects That Derive from the *View* Object . . . . . . . . . . . . . . . . . . . . . . . . . .325
Adding or Removing a View Programmatically . . . . . . . . . . . . . . . . . . . . . . .326
Customizing Your View . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .327
Specifying Fields in a View . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .327
Filtering Items in the *View* Object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .329
Sorting Items in a View . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .329
The *AutoFormatRules* Collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .330
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .334

**11 Searching Outlook Data. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 335**

Overview of Searching Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .335
Outlook Query Languages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .335
AQS. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .337
DASL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .342
Date-Time Comparisons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .354
Filtering Recurring Items in the Calendar Folder . . . . . . . . . . . . . . . . . . . . . .354
Date-Time Format of Comparison Strings . . . . . . . . . . . . . . . . . . . . . . . . . . . .355
Time Zones Used in Comparison. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .356
Conversion to UTC for DASL Queries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .357
Integer Comparisons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .358
Invalid Properties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .359
Jet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .359
DASL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .359
Comparison and Logical Operators. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .360
Comparison Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .360
Logical Operators. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .360
Null Comparisons. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .361
Search Entry Points. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .361
Search Considerations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .364
Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .364
Read-Only vs. Read/Write. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .365
Searching Subfolders . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .366
Windows Desktop Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .366
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .367

# Part IV Providing a User Interface for Your Solution


**12 Introducing the Outlook User Interface . . . . . . . . . . . . . . . . . . . . . . . . . 371**

Decoding the User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 371
The Explorer Window (The *Explorer* Object) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 372
Programming the *Explorer* Object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 373
The *Explorers* Collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 373
The Inspector Window (The *Inspector* Object) . . . . . . . . . . . . . . . . . . . . . . . . . . . 377
Programming the *Inspectors* Collection and *Inspector* Object . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 378
The *Inspectors* Collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 378
Working with the Navigation Pane. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 380
Making the Most of Navigation Modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . 380
Adding Structure with Navigation Groups . . . . . . . . . . . . . . . . . . . . . . . . . . . 382
Removing Folders . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 384
Folder Views . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 385
The Reading Pane . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 385
Customizing the Reading Pane . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 385
The To-Do Bar. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 386
Command Bars. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 386
Context Menus. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 386
Folder Home Pages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 389
Summary. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 390

**13 Creating Form Regions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 391**

Introduction to Form Regions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 391
Form Pages Compared with Form Regions. . . . . . . . . . . . . . . . . . . . . . . . . . . 392
Form Region Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 392
Standard Form Types. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 395
Anatomy of a Form Region Solution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 396
Becoming Familiar with Form Region Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . 396
Designing a Form Region. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 397
Adding Controls. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 399
Working with Fields. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 403
Polishing Your Form Region. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 406
Form Region End to End. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 411
Step 1: Creating a Form Region. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 411
Step 2: Writing Business Logic. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 415

Step 3: Registering the Form Region. . . . . 423
Advanced Form Region Methods . . . . . 433
Summary . . . . . 434

**14 Form Region Controls . . . . . 435**

Standard Controls. . . . . 435
The Outlook Check Box. . . . . 435
The Outlook Combo Box . . . . . 435
The Outlook Command Button . . . . . 436
The Outlook Label Control . . . . . 437
The Outlook List Box . . . . . 437
The Outlook Option Button . . . . . 437
The Outlook Text Box. . . . . 438
Outlook-Specific Controls . . . . . 438
The Outlook Body Control . . . . . 438
The Outlook Business Card Control. . . . . 438
The Outlook Category Control. . . . . 439
The Outlook Contact Photo Control . . . . . 440
The Outlook Date Control . . . . . 441
The Outlook Frame Header Control . . . . . 441
The Outlook InfoBar Control . . . . . 442
The Outlook Page Control . . . . . 443
The Outlook Recipient Control . . . . . 444
The Outlook Sender Photo Control. . . . . 444
The Outlook Time Zone Control . . . . . 445
The Outlook Time Control . . . . . 446
The Outlook View Control . . . . . 447
Using Form Region Controls. . . . . 447
Adding Controls to the Control Toolbox . . . . . 447
Adding Controls Programmatically . . . . . 448
Programmatic Access to Controls . . . . . 450
Summary . . . . . 452

**15 Extending the Ribbon. . . . . 453**

Introducing Ribbon Extensibility . . . . . 453
What Happens with Existing Code . . . . . 454
Outlook RibbonX Sample Add-In . . . . . 458
Installation Instructions . . . . . 458

Running the Sample Add-In . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 459
Modifying Your Code to Use RibbonX . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 459
Authoring Ribbon XML . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 461
*IRibbonExtensibility* Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 462
Detecting Errors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 465
*NewInspector* Event . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 466
*OutlookInspector* Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 467
*IRibbonUI* Object . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 468
*IRibbonControl* Object . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 468
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 470

**16 Completing Your User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 471**

Custom Task Panes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 471
When to Use a Custom Task Pane . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 472
Implementing a Custom Task Pane . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 472
Adding a Custom Task Pane in an Add-In . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 475
Custom Property Pages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 478
Designing a Custom Property Page . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 479
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 486

## Part V Advanced Topics

**17 Using the *PropertyAccessor* Object . . . . . . . . . . . . . . . . . . . . . . . . . . . 489**

Scenarios for *PropertyAccessor* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 489
Objects That Implement *PropertyAccessor* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 490
*PropertyAccessor* Namespaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 491
Obtaining a Specific *SchemaName* String . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 491
Type Specifiers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 492
The *Proptag* Namespace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 492
Named Property *ID* Namespace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 493
Named Property *String* Namespace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 494
*Office* Namespaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 495
DAV Namespaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 496
The *PropertyAccessor* Object . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 497
The *GetProperty* Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 497
The *SetProperty* Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 498
The *GetProperties* Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 499
The *SetProperties* Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 500

The *DeleteProperty* Method . . . . . . . . . . 501
The *DeleteProperties* Method . . . . . . . . . . 501
Date-Time Properties . . . . . . . . . . 502
Multivalued Properties . . . . . . . . . . 502
Helper Methods . . . . . . . . . . 503
Detecting and Reporting Error Conditions. . . . . . . . . . 505
Property Size Limitations. . . . . . . . . . 506
Summary . . . . . . . . . . 507

**18 Add-in Setup and Deployment. . . . . . . . . . 509**

Creating a Setup Project . . . . . . . . . . 509
Writing Required Keys to the Windows Registry . . . . . . . . . . 510
Installing to HKEY_CURRENT_USER . . . . . . . . . . 510
Installing to HKEY_LOCAL_MACHINE . . . . . . . . . . 510
Registry Keys Required for an Add-In. . . . . . . . . . 510
Registry Keys Required for a Form Region. . . . . . . . . . 512
Required Installation Components . . . . . . . . . . 512
.NET Framework Version 2.0. . . . . . . . . . 512
Visual Studio Tools for Office Runtime . . . . . . . . . . 513
Primary Interop Assemblies . . . . . . . . . . 514
Add-in Assembly and Other Required Components . . . . . . . . . . 516
Using a COM Shim. . . . . . . . . . 516
Writing Custom Actions . . . . . . . . . . 516
Deploying to Users Who Are Not Administrators. . . . . . . . . . 517
Summary . . . . . . . . . . 517

**19 Trust and Security . . . . . . . . . . 519**

Code Security for Outlook 2007. . . . . . . . . . 519
Guard Principles . . . . . . . . . . 522
Security Warning Types. . . . . . . . . . 523
Detecting Trusted State . . . . . . . . . . 525
Trapping Errors . . . . . . . . . . 526
Restricted Properties and Methods. . . . . . . . . . 526
Trusting Managed Code . . . . . . . . . . 531
Trustable Shared Add-Ins . . . . . . . . . . 531
Trust Center . . . . . . . . . . 532
Administrative Options . . . . . . . . . . 535
Group Policy Security for COM Add-Ins. . . . . . . . . . 535

Exchange-Brokered Security for COM Add-Ins . . . . . . . . . . . . . . . . . . . . . . . . 536
Configuring a Security Policy. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 536
Trusting an Add-In. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 537
Form Region Policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 540
Folder Home Page Policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 541
Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 542

**Index. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .543**

**What do you think of this book? We want to hear from you!**

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

**www.microsoft.com/learning/booksurvey/**

Chapter 10

# Organizing Outlook Data

When you complete this chapter, you should have a good understanding of the following areas:

- Overview of organizing information in Microsoft Office Outlook 2007
- Using categories and task flagging
- Creating rules programmatically
- Writing code to create a search folder
- Customizing views with new View objects

## How Outlook 2007 Helps to Organize Information

Outlook 2007 introduces several new features that help to organize the constantly growing number of items that arrive in a user's Inbox. Category colors and task flagging are easy to use and provide a simple tool for getting organized. Organizational schemes for mailbox items are almost as varied as the number of Outlook users. The focus of this chapter is not to prescribe the best method of organizing Outlook data. Rather, you'll learn how to use the Outlook object model to implement organizational schemes programmatically. The good news for developers is that the Outlook object model supports all the new organizational features of Outlook 2007. By writing a few lines of code, you can add color categories to items, mark items for follow-up, create rules, build custom search folders, or add views.

## The *Categories* Collection and *Category* Objects

Outlook 2007 provides color categorization functionality in which Outlook items can be categorized and displayed by category. Multiple color categories can be applied to a single Outlook item, and Outlook items can be grouped or sorted by color category. Shortcut keys can be assigned to each color category to allow users to more easily categorize items. Color categories are user defined, and can be created, deleted, and changed either programmatically or by user action within the Outlook user interface.

The *Category* object represents a single user-defined color category in the master category list, the list of color categories presented in the Outlook user interface and represented by the *Categories* collection of the *NameSpace* object. Unlike previous versions of Outlook, Outlook 2007 stores the master category list in the default store so that it will roam by default in most scenarios, as is the case with an Exchange mailbox. *Category* objects are identified with a globally unique identifier (GUID) when created, and this identifier cannot be changed. However,

you can change the name, color, and shortcut key associated with a color category by setting the *Name*, *Color*, and *ShortcutKey* properties of the *Category* object. The *CategoryID* property can be used to retrieve the identifier of a *Category* object.

Outlook items are displayed based on the category name stored in the *Categories* property of that Outlook item. The *Categories* property gets or sets a comma-delimited string of category names. It does not return a *Categories* collection object. Because category names are stored as part of the Outlook item, it is possible to add a category to an Outlook item that is not present in the master category list. For example, a category might have been removed. To determine if a category exists in the master category list, use the following *CategoryExists* method:

```
private bool CategoryExists(string categoryName)
{
    try
    {
        Outlook.Category category =
            Application.Session.Categories[categoryName];
        if (category != null)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    catch { return false; }
}
```

**Note** If the *Categories* property of an item contains a category name that does not exist in the *Categories* collection of the *Namespace* object, then the category name associated with that Outlook item is displayed, but without an associated color.

The following code sample enumerates the *Category* objects in the *Categories* collection and writes the *Name* and *CategoryID* properties to the trace listeners in the *Listeners* collection:

```
private void EnumerateCategories()
{
    Outlook.Categories categories =
        Application.Session.Categories;
    foreach (Outlook.Category category in categories)
    {
        Debug.WriteLine(category.Name);
        Debug.WriteLine(category.CategoryID);
    }
}
```

### Category Colors

The *Category* object exposes a *Color* property that lets you set or get an *olCategoryColor* constant. If you need to reproduce the color in a custom control, you can use these read-only properties of the *Category* object:

- *CategoryBorderColor*
- *CategoryGradientBottomColor*
- *CategoryGradientTopColor*

These properties return an OLE_COLOR value, which is dependent on the *Color* property of the *Category* object. For an advanced example of how to use *CategoryBorderColor*, *CategoryGradientBottomColor*, and *CategoryGradientTopColor*, see *ColorSwatchBuilder.cs* or *ColorSwatchBuilder.vb* in the PrepareMe sample add-in that accompanies this book.

## Creating a Category

To create a category programmatically, you call the *Add* method of the *Categories* collection. If the ISV category does not exist, the following code sample adds a category named ISV to the master category list and assigns the dark blue color to this category. It also assigns Ctrl+F11 as the shortcut key for the category.

```
private void AddACategory()
{
    Outlook.Categories categories =
        Application.Session.Categories;
    if(!CategoryExists("ISV"))
    {
        Outlook.Category category = categories.Add("ISV",
            Outlook.OlCategoryColor.olCategoryColorDarkBlue,
            Outlook.OlCategoryShortcutKey.olCategoryShortcutKeyCtrlF11);
    }
}
```

## Assigning One or More Categories to an Item

To assign categories to an item, use the *Categories* property on the item. The *Categories* property gets or sets a comma-delimited string that contains all of the categories assigned to the item. This property can contain a maximum of 255 characters, including the commas and spaces, to separate the category values. If you assign a category that is not in the *Categories* collection of the *Namespace* object, that category will not display a color. The following code sample creates a restriction for items that contain “ISV” in the subject. This code sample uses a *for* loop and the *OutlookItem* class to assign the ISV category to any item in the Inbox that contains “ISV” in the subject. Notice that the code sample examines the string returned by

*item.Categories* to determine if the *Categories* property is empty or already has been assigned to the ISV category.

```
private void AssignCategories()
{
    string filter = "@SQL=" + "\"" + "urn:schemas:httpmail:subject"
        + "\"" + " ci_phrasematch 'ISV'";
    Outlook.Items items =
        Application.Session.GetDefaultFolder(
        Outlook.OlDefaultFolders.olFolderInbox).Items.Restrict(filter);
    for(int i = 1; i<=items.Count; i++)
    {
        OutlookItem item = new OutlookItem(items[i]);
        string existingCategories = item.Categories;
        if(String.IsNullOrEmpty(existingCategories))
        {
            item.Categories = "ISV";
        }
        else
        {
            if (item.Categories.Contains("ISV") == false)
            {
                item.Categories = existingCategories + ", ISV";
            }
        }
        item.Save();
    }
}
```

## Displaying the Categories Dialog Box

The Outlook object model also provides the *ShowCategoriesDialog* method on an item to display the Categories dialog box, shown in Figure 10-1. This dialog box lets the user pick one or more categories that are assigned to the item. The user can also create new categories or clear existing categories with this dialog box. In the following code sample from the sample RulesAddin project that accompanies this book, a dummy mail item is created and the *ShowCategoriesDialog* method is called on the item. In this case, the categories selected by the user are displayed in an edit box and used to create a categories rule.

```
private void cmdCategory_Click(object sender, EventArgs e)
{
    try
    {
        //Create a dummy MailItem and display Categories dialog box
        Outlook.MailItem oMail = (Outlook.MailItem)m_olApp.CreateItem(
            Outlook.OlItemType.olMailItem);
        if (!string.IsNullOrEmpty(txtCategory.Text))
        {
            oMail.Categories = txtCategory.Text;
        }
        oMail.ShowCategoriesDialog();
        Application.DoEvents();
```

```
            if (!string.IsNullOrEmpty(oMail.Categories))
            {
                txtCategory.Text = oMail.Categories;
                chkCategory.Checked = true;
            }
            oMail = null;
        }
        catch(Exception ex)
        {
            LogMessage("cmdCategory_Click: "
                + ex.ToString() , EventLogEntryType.Error);
        }
    }
```

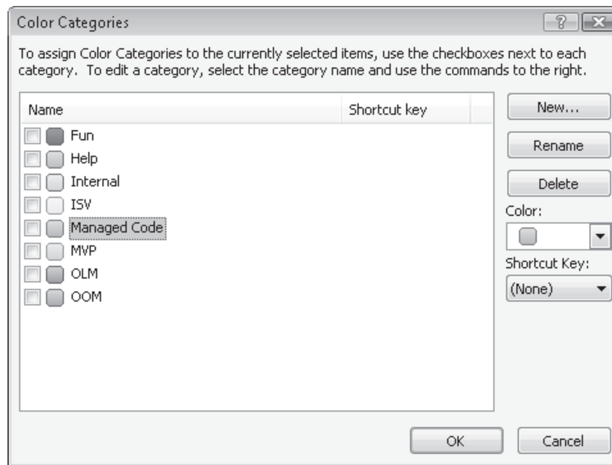


**Figure 10-1** Display the Categories dialog box programmatically.

# Task Flagging

Outlook 2007 provides a new task flagging system in which certain Outlook items such as mail items or contact items can be flagged for follow-up. Flagging an Outlook item for follow-up displays information about that Outlook item, along with other task-based information, on the To-Do Bar and Calendar navigation module in the Outlook user interface.

## Controlling Visibility of the To-Do Bar

The To-Do Bar is displayed as a vertical pane in a typical configuration of the Outlook Explorer window. It contains a date navigator control, upcoming appointments, and items that have been flagged for follow-up. The To-Do Bar itself is not extensible, and configuration options for the To-Do Bar can only be set through the Outlook user interface. You can programmatically change the visibility of the To-Do Bar using the *ShowPane* method of the *Explorer* object.

## Creating To-Do Items That Appear in the To-Do Bar

Creating to-do items programmatically is covered in the section “Create a To-Do Item” in Chapter 5, “Built-in Item Types.” Any item that is flagged for follow-up will appear in the To-Do Bar. As an organizational technique, item flagging creates a well-defined scheme for prioritizing tasks and to-do items. You should understand how to mark a group of items for a specified follow-up interval. The following code example processes all items in the user’s Inbox that are from the user’s manager and flags all high-importance items for follow-up today. If the item’s importance is normal, then the item is flagged for follow-up this week.

```
private void DemoTaskFlagging()
{
    const string PR_SENT_REPRESENTING_NAME =
        "http://schemas.microsoft.com/mapi/proptag/0x0042001E";
    const string PR_MESSAGE_CLASS =
        "http://schemas.microsoft.com/mapi/proptag/0x001A001E";
    Outlook.AddressEntry currentUser =
        Application.Session.CurrentUser.AddressEntry;
    if (currentUser.Type == "EX")
    {
        Outlook.ExchangeUser manager;
        try
        {
            manager = currentUser.
                GetExchangeUser().GetExchangeUserManager();
        }
        catch
        {
            Debug.WriteLine("Could not obtain user's manager.");
            return;
        }
        if (manager != null)
        {
            string displayName = manager.Name;
            string filter = "@SQL=" + "\""
            + PR_SENT_REPRESENTING_NAME + "\""
            + " = '" + displayName + "'" + " AND " + "\""
            + PR_MESSAGE_CLASS + "\"" + " = 'IPM.NOTE'";
            Outlook.Items items =
                Application.Session.GetDefaultFolder(
                Outlook.OlDefaultFolders.olFolderInbox).
                Items.Restrict(filter);
            foreach(Outlook.MailItem mail in items)
            {
                if (mail.Importance ==
                    Outlook.OlImportance.olImportanceHigh)
                {
                    mail.MarkAsTask(
                        Outlook.OlMarkInterval.olMarkToday);
                    mail.Save();
                }
                if (mail.Importance ==
                    Outlook.OlImportance.olImportanceNormal)
                {
```

```
                    mail.MarkAsTask(
                        Outlook.OlMarkInterval.olMarkThisWeek);
                    mail.Save();
                }
            }
        }
    }
}
```

# The *Rules* Collection and *Rule* Objects

Because they can operate either server-side or client-side, depending on the type of account and rule, Outlook rules provide one of the most powerful Outlook features for organizing information in a user's mailbox. Users implement rules to enforce their own organizational schemes. For example, some users like to create a hive of subfolders that contain unread mail and read mail by subject area. Other users might create a subfolder hierarchy that corresponds to the sender of the message. Still others categorize their mail and then use search folders to aggregate the mail by category. As stated at the beginning of this chapter, users follow a multiplicity of schemes when they organize the items in their mailboxes. The new Rules object model in Outlook 2007 allows you as a developer to participate in the power of rules. You can create rules programmatically to enforce a certain organizational scheme, create a specific rule that is unique to your solution, or ensure that certain rules are deployed to a group of users.

The Rules object model supports the programmatic adding, editing, and deleting of rules. The *Rules* collection and *Rule* objects allow you to access, add, and delete rules defined for a session. The *RuleAction* and *RuleCondition* objects, their collection objects, and derived action and condition objects further support editing actions and conditions.

**Note** The Rules object model provides partial parity with the Rules and Alerts Wizard in the Outlook user interface. Although it does not support every single rule that you can possibly create using the wizard, it supports the most commonly used rule actions and conditions. Just like any rule created using the Rules and Alerts Wizard, rules created programmatically are applied to messages, which include mail items, meeting requests, task requests, documents, delivery receipts, read receipts, voting responses, and out-of-office notices.

## Overview of Rules Programming

Creating one or more rules programmatically is straightforward once you understand the architecture of the Rules object model. Figure 10-2 illustrates the basic architecture of the Rules object model. Note that there is no separate collection that represents rule exception conditions. Rule exception conditions are accessed through the *Exceptions* property of the *Rule* object. The *Exceptions* property returns a *RuleCollections* object.

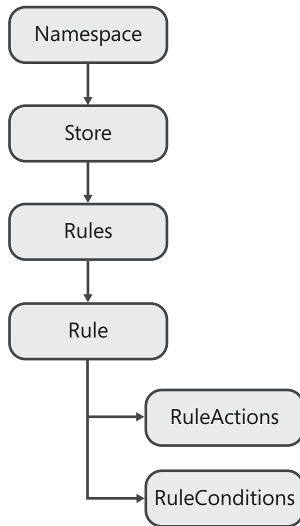


**Figure 10-2** Rules object model architecture.

Now that you understand the architecture of the *Rules* objects, it's time to move on to practical coding instructions. From a top-level perspective, follow these steps when you create rules through the Outlook object model:

1. Obtain the *Rules* collection from the *DefaultStore* property of the *Namespace* object. Call the *GetRules* method on *DefaultStore* to obtain the *Rules* collection. You should write this code in a *try...catch* block because Outlook will raise an error if the user is offline or disconnected from the Exchange server.
2. Call the *Create* method on the *Rules* object to create an instance variable for a *Rule* object. When you call the *Create* method, you specify a *Name* and a *RuleType* parameter. *RuleType* determines whether the *Rule* object is a send or receive rule. Send rules operate on outgoing messages and receive rules operate on incoming messages. You cannot change the *RuleType* property after the *Rule* object has been created. If you apply inappropriate conditions to a *Rule* instance (such as a *NewItemAlert* action to a send rule), Outlook raises an error when you call the *Save* method on the *Rules* collection.
3. Use the *RulesActions* and *RuleConditions* collections to enable actions, conditions, and exceptions on the *Rule* object. Note that the *Exceptions* property on a *Rule* object returns a *RuleConditions* collection, and any condition enabled in this collection is treated as a rule exception condition. These collection objects represent static collections, meaning that you cannot add additional built-in or custom actions or conditions to the collection.
4. For any given *Rule* action, condition, or exception to be operational, you must first set its *Enabled* property to *true*. For some actions or conditions, this is all that you have to do. For other actions or conditions, such as the *MoveOrCopyRuleAction.Folder* property, you must set additional properties on the action or condition to save the *Rule* object without an error.

5. Finally, you call the *Save* method on the *Rules* collection to persist the created or modified rules to storage. Again, it is recommended that you enclose the *Save* method in a *try...catch* block to handle exceptions.

Next you'll see a detailed code sample that implements the steps just described. If the *CurrentUser* property represents an *ExchangeUser* object, the *CreateManagerRule* procedure obtains the *ExchangeUser* object for the manager of the *CurrentUser* property of the *Namespace* object. The *Rules* object model is used to create a receive rule that moves received messages to a subfolder of the Inbox if the message is from the user's manager, the recipient is on the To line of the message, and the message is not a meeting request or update. Additionally, the message is marked for follow-up today.

Although this code sample is extensive, it provides you with a great start for understanding how to use the Rules object model. It also illustrates appropriate error handling for conditions that could raise an exception under certain conditions such as the user being offline or disconnected in cached Exchange mode. As you read through the code, notice that each of the steps discussed earlier has been implemented in the code sample.

```
private void CreateManagerRule()
{
    Outlook.ExchangeUser manager;
    Outlook.Folder managerFolder;
    Outlook.AddressEntry currentUser =
        Application.Session.CurrentUser.AddressEntry;
    if (currentUser.Type == "EX")
    {
        try
        {
            manager = currentUser.
                GetExchangeUser().GetExchangeUserManager();
        }
        catch
        {
            Debug.WriteLine("Could not obtain user's manager.");
            return;
        }
        Outlook.Rules rules;
        try
        {
            rules = Application.Session.DefaultStore.GetRules();
        }
        catch
        {
            Debug.WriteLine("Could not obtain rules collection.");
            return;
        }
        if (manager != null)
        {
            string displayName = manager.Name;
            Outlook.Folders folders =
                Application.Session.GetDefaultFolder(
                Outlook.OlDefaultFolders.olFolderInbox).Folders;
```

```
            try
            {
                managerFolder =
                    folders[displayName] as Outlook.Folder;
            }
            catch
            {
                managerFolder =
                    folders.Add(displayName, Type.Missing)
                    as Outlook.Folder;
            }
            Outlook.Rule rule = rules.Create(displayName,
                Outlook.OlRuleType.olRuleReceive);
            //Rule conditions
            //From condition
            rule.Conditions.From.Recipients.Add(
                manager.PrimarySmtpAddress);
            rule.Conditions.From.Recipients.ResolveAll();
            rule.Conditions.From.Enabled = true;
            //Sent only to me
            rule.Conditions.ToMe.Enabled = true;
            //Rule exceptions
            //Meeting invite or update
            rule.Exceptions.MeetingInviteOrUpdate.Enabled = true;
            //Rule actions
            //MarkAsTask action
            rule.Actions.MarkAsTask.MarkInterval =
                Outlook.OlMarkInterval.olMarkToday;
            rule.Actions.MarkAsTask.FlagTo = "Follow-up";
            rule.Actions.MarkAsTask.Enabled = true;
            //MoveToFolder action
            rule.Actions.MoveToFolder.Folder = managerFolder;
            rule.Actions.MoveToFolder.Enabled = true;
            try
            {
                rules.Save(true);
            }
            catch(Exception ex)
            {
                Debug.WriteLine(ex.Message);
            }
        }
    }
}
```

## *Rules* Collection

The *Rules* collection represents a set of *Rule* objects that are the rules available in the current session.

## Obtaining the *Rules* Collection

To obtain the *Rules* collection, you call the *GetRules* method on the *DefaultStore* property of the *Namespace* object. For users connected to an Exchange server, calling *GetRules* can be an expensive operation in terms of performance on slow connections.

The order of the *Rule* objects in the collection returned from *GetRules* follows that of *Rule.ExecutionOrder*, with *ExecutionOrder* equal to 1 being the first *Rule* object in the collection and *Rule.ExecutionOrder* equal to *Rules.Count* being the last *Rule* object in the collection.

**Tip** You should scope the lifetime of the *Rules* collection to the most constrained possible scope. Outlook enforces "last writer wins" when the *Rules* collection is saved. If another add-in or the Rules and Alerts Wizard modifies rules while your add-in holds onto an instance of the *Rules* collection, you might see unexpected results after you call *Rules.Save*.

## Creating a *Rule* Object

To create an instance of a *Rule* object, call the *Create* method on the *Rules* collection. Depending on whether you want to create a send rule or a receive rule, specify an appropriate *OlRuleType* constant to the *Create* method. The *RuleType* parameter of the added rule determines valid rule actions, rule conditions, and rule exception conditions that can be associated with the *Rule* object. Newly created rules are enabled by default. If you want to create the rule and also leave it disabled, you must explicitly set its *Enabled* property to *false*. When a rule is added to the collection, the *Rule.ExecutionOrder* value of the new rule is 1. The *ExecutionOrder* value of other rules in the collection is incremented by 1. The newly created *Rule* object is not persisted until you call the *Save* method on the *Rules* collection. However, you can call the *Execute* method on the *Rule* object before you save the collection.

## Enumerating Rules

Use the Indexer to enumerate rules in the *Rules* collection. Once you have obtained a *Rule* object, you can enable or disable the rule by changing its *Enabled* property. You can also modify the existing rule actions, conditions, and exceptions. Finally, you can execute the rule by calling the *Execute* method on the *Rule* object. The following code sample enumerates all the rules in the *Rules* collection and writes the rule's *Name*, *IsLocalRule,* and *Enabled* properties to the trace listeners in the *Listeners* collection:

```
private void EnumerateRules()
{
    Outlook.Rules rules =
        Application.Session.DefaultStore.GetRules();
    foreach (Outlook.Rule rule in rules)
    {
        StringBuilder sb = new StringBuilder();
        sb.AppendLine("Name: "
```

```
            + rule.Name);
        sb.AppendLine("Local: "
            + rule.IsLocalRule.ToString());
        sb.AppendLine("Enabled: "
            + rule.Enabled.ToString());
        Debug.WriteLine(sb.ToString());
    }
}
```

**Note** You can retrieve each rule in a *Rules* collection by indexing the collection using *Rules[Index]*, with *Index* being either the name of the rule (the default property *Rule.Name*), or a value ranging from 1 through the total number of rules in the collection, *Rules.Count.Rule.ExecutionOrder* indicates the order of execution of the rules in the collection and is directly mapped with the numerical value of *Index* in *Rules[Index]*. For example, *Rules[1]* represents a rule with *Rule.ExecutionOrder* being 1, *Rules[2]* represents a rule with *Rule.ExecutionOrder* being 2, and *Rules[Rules.Count]* represents the rule with *Rule.ExecutionOrder* being *Rules.Count*.

## RSS Rules Processing

The *Rules* collection exposes an *IsRssRulesProcessingEnabled* property that controls whether RSS rule conditions are evaluated for RSS items. To persist changes to this property, you must call *Rules.Save*. The *IsRssRulesProcessingEnabled* property corresponds to the Enable Rules On All RSS Feeds check box in the Rules And Alerts dialog box, shown in Figure 10-3.

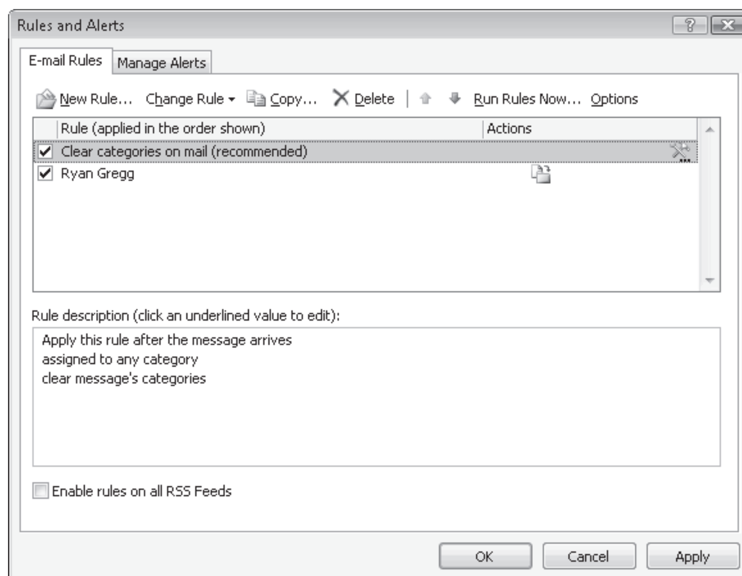


**Figure 10-3** Rules And Alerts dialog box.

If you set *IsRssRulesProcessingEnabled* to *true,* you can create RSS rules that operate in a different manner than default RSS rules that move RSS items from a specific feed to a subfolder of the RSS Subscriptions folder. If *IsRssRulesProcessingEnabled* is *false*, then no conditions about RSS feeds will be evaluated during rules processing. To create a rule that operates on RSS items, enable the *FromRssFeed* or *FromAnyRssFeed* rule conditions.

## Deleting a Rule

To delete a *Rule*, call the *Remove* method on the *Rules* collection. *Rules.Remove* removes from the *Rules* collection a *Rule* object specified by *Index*, which is either a numerical index into the *Rules* collection or the rule name. You must call *Rules.Save* to persist the deletion.

## Saving Rules

You do not save an individual *Rule* object. Instead you must call the *Save* method on the *Rules* collection to save all the *Rule* objects in the collection. After you enable a rule, you must also save the rule by using *Rules.Save* so that the rule and its enabled state will persist beyond the current session. A rule is only enabled after it has been saved successfully.

If you set the *ShowProgress* argument of the *Save* method to *true*, Outlook displays a progress dialog box. If you are saving rules on a slow connection to an Exchange server, *Rules.Save* is an expensive operation in terms of performance. In this circumstance it is advisable to display the progress dialog box; otherwise the user might believe that Outlook has hung.

## Handling Errors During a Save Operation

Always place *Rules.Save* in a *try...catch* construct. The connection to the Exchange server can go down, and you must be able to handle this exception. Exchange Server limits the maximum number of rules that can be supported by a store. The rules limit depends on the version of Exchange Server. For Microsoft Exchange Server 2007, an Exchange administrator can also control the rules limit per mailbox. *Rules.Save* returns an error when this limit is reached. The limit is generally not an issue for users running against a Post Office Protocol 3 (POP3) or Internet Message Access Protocol (IMAP) account, because all rules operate locally.

Saving rules that are incompatible or have improperly defined actions or conditions (such as an empty string for *TextRuleCondition.Text* or *MarkAsTaskRuleAction.FlagTo*) will return an error. Some combinations of *RuleActions* and *RuleConditions* are incompatible and will also return an error.

If an error occurs during *Rules.Save,* the entire save operation is rolled back. Modified rules are not saved and newly created or deleted rules are discarded. Unfortunately, the error that bubbles up to your code will not tell you exactly which rules or combination of *RuleActions* and *RuleConditions* caused the error to occur.

# The *Rule* Object

The *Rule* object represents an Outlook rule. A *Rule* object has a *RuleType* property that indicates whether the rule is a send or receive rule. *RuleType* is specified when the rule is created. *RuleType* cannot be changed without deleting the rule and re-creating the rule with a different *RuleType* property.

A rule can execute on the Exchange server or on the Outlook client, provided that the current user's mailbox is hosted on an Exchange server. If the rule executes on the server, Outlook does not have to be running for the rule conditions to be evaluated and the rule actions to be completed. If the rule executes on the client, meaning that the *IsLocalRule* property of the *Rule* object returns *true*, then Outlook must be running for the rule to execute.

## Executing a Rule

To cause a rule to execute immediately, call the *Execute* method on the *Rule* object. Use *Rule.Execute* to apply a rule as a one-off operation regardless of whether *Rule.Enabled* returns *true*. Use *Rule.Enabled* and then *Rules.Save* if you want to apply the rule consistently and persist the rules beyond the current session. The following code sample executes the rule created in the *CreateManagerRule* procedure shown earlier:

```
private void ExecuteManagerRule()
{
    Outlook.AddressEntry currentUser =
        Application.Session.CurrentUser.AddressEntry;
    if (currentUser.Type == "EX")
    {
        try
        {
            string managerName = currentUser.
                GetExchangeUser().GetExchangeUserManager().Name;
            Outlook.Rule managerRule =
                Application.Session.DefaultStore.GetRules()[managerName];
            if (managerRule != null)
            {
                managerRule.Execute(false, Type.Missing,
                    Type.Missing, Type.Missing);
            }
        }
        catch(Exception ex)
        {
            Debug.WriteLine(ex.Message);
        }
    }
}
```

The parameters to the *Execute* method are optional. If you do not specify any parameters, the rule will be applied to all messages in the Inbox but not to the subfolders of the Inbox. The default values for the optional arguments for the *Execute* method are shown in Table 10-1.

**Table 10-1 Parameters for *Rules.Execute***

| Parameter | Default value |
|---|---|
| ShowProgress | False |
| Folder | Inbox |
| IncludeSubfolders | False |
| RuleExecuteOption | OlRuleExecuteOption.olRuleExecuteAllMessages |

If *ShowProgress* is *true* and the user cancels the progress dialog box, rule execution is canceled in the same manner as if the user had canceled rule execution through the Rules and Alerts Wizard. *Execute* returns an error when the user cancels the progress dialog box.

If you plan to show a custom progress user interface instead of using the progress dialog box, you should be aware that there are no events that indicate when rule execution starts and stops.

## Causing a Rule to Operate Locally

To cause a server-side rule to operate locally, enable the *OnLocalMachine* rule condition. For some rule actions that must run on the client (such as displaying a new mail alert or playing a sound), the *OnLocalMachine* condition will be enabled by default when you set the *Enabled* property to *true* for a client-side only *RuleAction* object. For other rule actions that normally run on the server, you can enable an *OnLocalMachine* condition that will force the rule to run locally on the client. The following code sample illustrates how an *OnLocalMachine* condition forces a server-side rule to run locally. Normally a *Forward* action and *OnlyToMe* condition will operate on the server. In this case they operate as a client-side rule because the *OnLocalMachine* condition has been enabled.

```
private void DemoOnMachineOnly()
{
    Outlook.Rules rules =
        Application.Session.DefaultStore.GetRules();
    Outlook.Rule rule =
        rules.Create("Demo Machine Only Rule",
        Outlook.OlRuleType.olRuleReceive);
    rule.Conditions.OnlyToMe.Enabled = true;
    rule.Actions.Forward.Enabled = true;
    rule.Actions.Forward.Recipients.Add("someone@example.com");
    rule.Actions.Forward.Recipients.ResolveAll();
    //Force the rule to execute locally
    rule.Conditions.OnLocalMachine.Enabled = true;
    rules.Save(true);
}
```

**Note** The corollary of enabling the *OnLocalMachine* condition for a rule is that the *OnOtherMachine* condition will be enabled when the same rule is examined from another machine. You cannot programmatically enable or disable a condition of type *olConditionOtherMachine*. This type of rule condition indicates that the rule can run only on a specific computer that is not the current one. This happens when the rule is created on that computer and the *OnLocalMachine* rule condition is enabled, indicating that the rule can run only on that computer. When you run the same rule on another computer, the rule will show that the *OnOtherMachine* rule condition is enabled.

## The *RuleActions* Collection

The *RuleActions* collection contains a set of *RuleAction* objects or objects derived from *RuleAction*, representing the actions that are executed on a *Rule* object. The actions exposed on the *RuleActions* collection let you enable or disable the action programmatically by setting the *Enabled* property of a given rule action. The number of rule actions in the *RuleActions* object is fixed.

Although the *RulesActions* collection lets you determine the rule actions that are enabled for a given *Rule* object, not all *RuleAction* objects are supported for programmatic creation of rule actions. For example, you cannot enable a rule action in your code that assigns the *Importance* property to an item. However, your code can recognize a rule action created through the Rules and Alerts Wizard that enables an action that assigns the *Importance* property. In this case, *RuleAction.ActionType* would return *OlRuleActionType.olRuleActionImportance*. You could write code similar to the following to determine that such a rule action exists. Note that you cannot determine the *Importance* value assigned by the rule action.

```
private void ParseImportanceRuleAction()
{
    Outlook.Rules rules =
        Application.Session.DefaultStore.GetRules();
    Outlook.Rule rule =
        rules["Importance Rule"];
    foreach (Outlook.RuleAction ruleAction in rule.Actions)
    {
        if (ruleAction.ActionType ==
            Outlook.OlRuleActionType.olRuleActionImportance)
        {
            Debug.WriteLine(ruleAction.Enabled.ToString());
        }
    }
}
```

Table 10-2 lists all rules actions listed by *OlRuleActionType*. From this table, you can determine which rule actions are supported when creating a rule programmatically by looking at the Valid When Creating New Rules with Code? column. You can also determine which rule actions are valid for receive and send rules.

Table 10-2 **Rule Actions by *OlRuleActionType***

| **Action** | **Constant in *OlRuleActionType*** | **Valid when creating new rules with code?** | **Valid for receive rules?** | **Valid for send rules?** |
|---|---|---|---|---|
| Assign the message to the categories specified in the *Categories* property. | olRuleActionAssignTo-Category | Yes | Yes | Yes |
| Cc the message to the recipient list specified in the *Recipients* property. | olRuleActionCcMessage | Yes | No | Yes |
| Clear all categories for the message. | olRuleActionClear-Categories | Yes | Yes | Yes |
| Copy the message to the folder specified in the *Folder* property. | olRuleActionCopyToFolder | Yes | Yes | Yes |
| Run a custom action. | olRuleActionCustomAction | No | Yes | Yes |
| Defer the delivery by a specified number of minutes. | olRuleActionDefer | No | No | Yes |
| Delete the message. | olRuleActionDelete | Yes | Yes | No |
| Permanently delete the message. | olRuleActionDelete-Permanently | Yes | Yes | No |
| Display a desktop alert. | olRuleActionDesktopAlert | Yes | Yes | No |
| Clear the message flag. | olRuleActionFlagClear | No | Yes | No |
| Flag the message with the color specified. | olRuleActionFlagColor | No | Yes | No |
| Flag the message for action in days specified. | olRuleActionFlagFor-ActionInDays | No | Yes | Yes |
| Forward the message to the recipient list specified in the *Recipients* property. | olRuleActionForward | Yes | Yes | No |
| Forward the message as an attachment to the recipient list specified in the *Recipients* property. | olRuleActionForwardAs-Attachment | Yes | Yes | No |
| Mark the message with the specified *Importance* value. | olRuleActionImportance | No | Yes | Yes |
| Mark message as a task for follow-up using the *FlagTo* and *MarkInterval* proper-ties of the *MarkAsTask-RuleAction* object. | olRuleActionMarkAsTask | Yes | Yes | No |
| Mark as read. | olRuleActionMarkRead | No | Yes | No |

Table 10-2 Rule Actions by *OlRuleActionType*

| Action | Constant in *OlRuleActionType* | Valid when creating new rules with code? | Valid for receive rules? | Valid for send rules? |
|---|---|---|---|---|
| Move the message to the folder specified in the *Folder* property. | olRuleActionMoveToFolder | Yes | Yes | No |
| Display the message specified in the *Text* property. | olRuleActionNewItemAlert | Yes | Yes | No |
| Notify that the message has been delivered. | olRuleActionNotifyDelivery | Yes | No | Yes |
| Notify that the message has been read. | olRuleActionNotifyRead | Yes | No | Yes |
| Play the .wav file specified in the *FilePath* property. | olRuleActionPlaysound | Yes | Yes | No |
| Print the message to the default printer. | olRuleActionPrint | No | Yes | No |
| Redirect the message to the recipient list specified in the *SendRuleAction.Recipients* property. | olRuleActionRedirect | Yes | Yes | No |
| Start a script. | olRuleActionRunScript | No | Yes | No |
| Mark the message with the specified sensitivity. | olRuleActionSensitivity | No | No | Yes |
| Have server reply using the specified message. | olRuleActionServerReply | No | Yes | No |
| Start an .exe file. | olRuleActionStartApplication | No | Yes | No |
| Stop processing more rules. | olRuleActionStop | Yes | Yes | Yes |
| Reply using the specified template (.oft) file. | olRuleActionTemplate | No | Yes | No |
| Unrecognized rule action. | olRuleActionUnknown | No | Yes | No |

# The *RuleConditions* Collection

The *RuleConditions* collection contains a set of *RuleCondition* objects or objects derived from *RuleCondition*, representing the conditions or exception conditions that must be satisfied for the *Rule* to execute. The conditions exposed on the *RuleConditions* collection let you enable or disable the condition programmatically by setting the *Enabled* property of a given rule condition. The number of rule conditions in the *RuleConditions* collection is fixed.

Although the *RuleConditions* collection lets you determine the rule conditions that are enabled for a given *Rule* object, not all *RuleCondition* objects are supported for programmatic creation

of rule conditions. See the earlier discussion of *RuleActions* for a method of determining which conditions are enabled for a given rule.

Table 10-3 lists all rules actions listed by *OlRuleConditionType*. From this table, you can determine which rule conditions are supported when creating a rule programmatically by looking at the Valid When Creating New Rules with Code? column. You can also determine which rule conditions are valid for receive and send rules.

**Table 10-3 Rule Actions by *OlRuleConditionType***

| Condition | Constant in *OlRuleConditionType* | Valid when creating new rules with code? | Valid for receive rules? | Valid for send rules? |
|---|---|---|---|---|
| Account is the account specified in the *Account* property. | olConditionAccount | Yes | Yes | Yes |
| Message is assigned any category. | olConditionAnyCategory | Yes | Yes | Yes |
| Body contains words specified in *Text* property. | olConditionBody | Yes | Yes | Yes |
| Body or subject contains words specified in *Text* property. | olConditionBodyOrSubject | Yes | Yes | Yes |
| Message is assigned the category or categories specified in the *Categories* property. | olConditionCategory | Yes | Yes | Yes |
| Message has my name in the Cc box. | olConditionCc | Yes | Yes | No |
| Message was received between x and y, where x and y are *Integer* values. | olConditionDateRange | No | Yes | Yes |
| Message is flagged for the specified action. | olConditionFlaggedForAction | No | Yes | Yes |
| Message uses the form specified in the *FormName* property. | olConditionFormName | Yes | Yes | Yes |
| Sender is in the recipient list specified in the *Recipients* property. | olConditionFrom | Yes | Yes | No |
| Message is generated from any RSS subscription. | olConditionFromAnyRssFeed | Yes | Yes | No |
| Message is generated from a specified RSS subscription. | olConditionFromRssFeed | Yes | Yes | No |

Table 10-3 **Rule Actions by *OlRuleConditionType***

| Condition | Constant in *OlRuleConditionType* | Valid when creating new rules with code? | Valid for receive rules? | Valid for send rules? |
|---|---|---|---|---|
| Message has an attachment. | olConditionHasAttachment | Yes | Yes | Yes |
| Message is marked with the specified level of importance. | olConditionImportance | Yes | Yes | Yes |
| Rule can run only on this machine. | olConditionLocalMachineOnly | Yes | Yes | Yes |
| Message is a meeting invitation or update. | olConditionMeetingInviteOrUpdate | Yes | Yes | Yes |
| Message header contains words specified in the *Text* property. | olConditionMessageHeader | Yes | Yes | No |
| Message does not have my name in the To box. | olConditionNotTo | Yes | Yes | No |
| Message is sent only to me. | olConditionOnlyToMe | Yes | Yes | No |
| Message is an out-of-office message. | olConditionOOF | No | Yes | No |
| Rule can run only on a specific machine that is not the current one. | olConditionOtherMachine | No | Yes | Yes |
| Document property is exactly, contains, or does not contain specified properties. | olConditionProperty | No | Yes | Yes |
| Recipient address contains words specified by the *Text* property. | olConditionRecipientAddress | Yes | Yes | Yes |
| Sender address contains words specified by the *Text* property. | olConditionSenderAddress | Yes | Yes | No |
| Sender is in the address list specified in the *Address* property. | olConditionSenderInAddressBook | Yes | Yes | No |
| Message is marked with the specified level of sensitivity. | olConditionSensitivity | No | Yes | Yes |
| Sent to recipients (To, Cc) are in the recipient list specified in the *Recipients* property. | olConditionSentTo | Yes | Yes | Yes |

**Table 10-3 Rule Actions by *OlRuleConditionType***

| Condition | Constant in *OlRuleConditionType* | Valid when creating new rules with code? | Valid for receive rules? | Valid for send rules? |
|---|---|---|---|---|
| Message size is between x and y in units of KB, where x and y are Date values. For example, "10;50" sets the size condition between 10 and 50KB. | olConditionSizeRange | No | Yes | Yes |
| Subject contains words specified in the *Text* property. | olConditionSubject | Yes | Yes | Yes |
| My name is in the To box. | olConditionTo | Yes | Yes | No |
| Message has my name in the To or Cc box. | olConditionToOrCc | Yes | Yes | No |
| Unrecognized rule condition. | olConditionUnknown | No | Yes | No |

## Get or Set Action or Condition Properties with an Array

Certain actions or conditions get or set an array that represents the conditions to be evaluated or the actions to be completed. The most notable example is the *Text* property of the *TextRuleCondition*. The *Text* property returns or sets an array of *string* elements that represents the text to be evaluated by the rule condition. For the *Text* property, you must assign an array with one string or multiple strings for evaluation. Multiple text strings assigned in an array are evaluated using the logical *OR* operation. Properties that get or set an array are as follows:

- *AddressRuleCondition.Address*
- *AssignToCategoryRuleAction.Categories*
- *CategoryRuleCondition.Categories*
- *FormNameRuleCondition.FormName*
- *TextRuleCondition.Text*

The following code sample shows you how to use arrays for some of these properties. In this sample, a rule is created that assigns categories based on conditional evaluation of the words "Office," "Outlook," and "2007" in the subject of the item. If the condition is satisfied, then the categories of Office and Outlook are assigned to the item. Note that the code checks for the existence of these categories in the *Categories* collection using the *CategoryExists* method listed earlier in this chapter. If the category does not exist, the category is added to the master category list.

```
private void CreateTextAndCategoryRule()
{
    if(!CategoryExists("Office"))
```

```
    {
        Application.Session.Categories.Add(
            "Office",Type.Missing, Type.Missing);
    }
    if(!CategoryExists("Outlook"))
    {
        Application.Session.Categories.Add(
            "Outlook",Type.Missing, Type.Missing);
    }
    Outlook.Rules rules =
        Application.Session.DefaultStore.GetRules();
    Outlook.Rule textRule =
        rules.Create("Demo Text and Category Rule",
        Outlook.OlRuleType.olRuleReceive);
    Object[] textCondition =
        { "Office", "Outlook", "2007" };
    Object[] categoryAction =
        { "Office", "Outlook" };
    textRule.Conditions.BodyOrSubject.Text =
        textCondition;
    textRule.Conditions.BodyOrSubject.Enabled = true;
    textRule.Actions.AssignToCategory.Categories =
        categoryAction;
    textRule.Actions.AssignToCategory.Enabled = true;
    rules.Save(true);
}
```

## Rules Sample Add-In

The Rules Sample add-in is available in a Microsoft Visual Basic .NET version (RulesAddinVB) and in a C# version (RulesAddinCS) in the sample code on this book's companion Web site.

The Rules Sample add-in demonstrates how you can substitute a custom Microsoft Windows Form dialog box for the default Outlook Create Rule dialog box that can be invoked from the context menu for an item. Corporate developers can modify and extend this example to create their own version of the Rules Sample add-in. The custom dialog box could promote the creation of rules that you want to deploy in your organization. Figure 10-4 shows the default Outlook Create Rule dialog box.

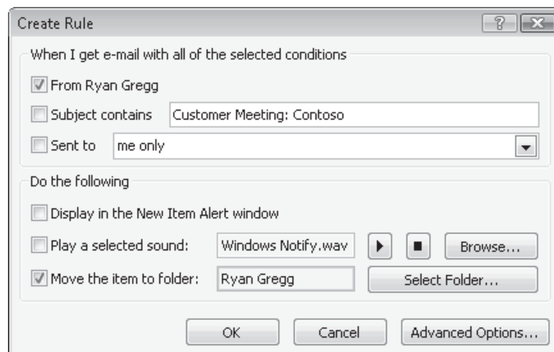


**Figure 10-4** Outlook Create Rule dialog box.

When you build and install the Rules Sample add-in following the instructions that accompany the sample, you'll find that the add-in has repurposed the *Create Rule* command on the item context menu so that the custom Windows Form dialog box, shown in Figure 10-5, appears in place of the default Outlook Create Rule dialog box. Due to space limitations, the Rules Sample add-in is not discussed in detail here. Although this sample is relatively simple, it is packed with great code samples for creating rules programmatically and repurposing command bar and *Ribbon* commands.

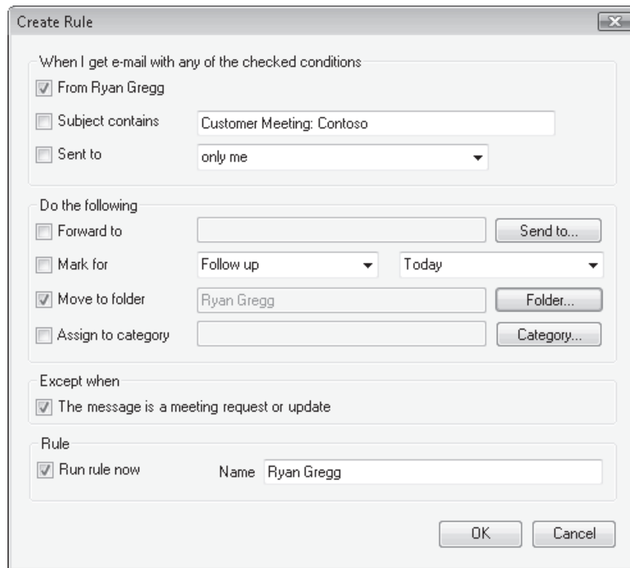


**Figure 10-5** Custom Windows Forms dialog box appears in place of the default Create Rules dialog box.

# Search Folders

Search folders provide another way to organize Outlook data. Think of a search folder as a virtual folder that can contain items located across different folders in a given store. This section shows you how to create search folders programmatically. You can create and persist search folders so that they are visible in the Outlook folder hierarchy, or you can create searches dynamically that are not saved. If the search folder is not saved, it will not appear in the folder hierarchy. If a search folder is an integral component of your solution, you should consider adding your solution search folder to the user's favorite folders to promote its visibility. Later in this chapter you'll see how you can create a search folder programmatically and add that search folder to the user's favorite folders.

## When to Use a Search Folder

A search folder provides a virtual folder that contains items that meet a set of search criteria. If you want to use a search folder in your solution, you should understand the following guidelines for search folders:

- Search folders are only supported for items in mail folders.
- You can run multiple searches simultaneously by calling the *AdvancedSearch* method in successive lines of code. A maximum of 100 simultaneous searches can be performed using the Microsoft Outlook user interface and the Outlook object model.
- You can only create the criteria for a search folder using a DAV Searching and Locating (DASL) query. For additional information on DASL and Jet query languages, see Chapter 11, "Searching Outlook Data." Note that you cannot use a Microsoft Jet query for the *Filter* parameter of *AdvancedSearch*. If Instant Search is enabled on a store that contains a folder specified in the *Scope* parameter, you can use Instant Search keywords to improve the performance of your search. If you use Instant Search keywords and Instant Search is not enabled, Outlook will return an error and your search will fail.
- Creating search folders on Exchange Server can affect the server's performance. For additional information on search folders and performance, see the section "Performance" in Chapter 11.
- Search folders can search in multiple folders and subfolders within a store. To specify multiple folders for the *Scope* parameter, use a comma character between each folder path and enclose each folder path in single quotes.
- The Outlook object model does not allow you to modify search folder criteria dynamically. If you create a search folder programmatically, the end user cannot modify criteria for the search folder. If you need to modify the criteria for a programmatically created search folder, you must delete the search folder programmatically and then re-create it. The end user can modify the scope for a programmatically created search folder, but it cannot be modified programmatically for an existing search folder.
- Use the *GetTable* method of the *Search* object or the *Search.Results* object to enumerate items returned by the search. When you obtain a *Table* object from the *GetTable* method, you can add or remove table columns. However, you cannot call the *Restrict* method on the *Table* object to modify the original criteria specified by the *Filter* parameter to *AdvancedSearch*.
- Because the results of *AdvancedSearch* can be returned asynchronously, you should use the *AdvancedSearchComplete* event of the *Application* object to obtain the results of the search. Use the *IsSynchronous* property of the *Search* object to determine if the search is synchronous or asynchronous.
- Search folders cannot span stores.
- Outlook 2007 does not support search folders for appointment, contact, task, and other folder types.

## Enumerating Search Folders

To enumerate search folders, you call the *GetSearchFolders* method on the *Store* object. *GetSearchFolders* returns all the visible active search folders for the *Store* object. It does not return uninitialized or aged-out search folders. *GetSearchFolders* returns a *Folders* collection object with

*Folders.Count* equal to zero (0) if no search folders have been defined for the store. Not all store providers (the Exchange public folder store, for example) support search folders. If the store provider does not support search folders, calling *Store.GetSearchFolders* will raise an error.

The following code sample enumerates the search folders on all .pst or .ost stores for the current session and writes the search folder path to the trace listeners in the *Listeners* collection:

```
private void EnumerateAllSearchFolders()
{
    Outlook.Stores stores = Application.Session.Stores;
    foreach (Outlook.Store store in stores)
    {
        if (store.IsDataFileStore)
        {
            Outlook.Folders folders = store.GetSearchFolders();
            foreach (Outlook.Folder folder in folders)
            {
                Debug.WriteLine(folder.FolderPath);
            }
        }
    }
}
```

**Note** Although you can enumerate search folders programmatically, you cannot activate a search folder using code. You also cannot determine the built-in or custom criteria for an existing search folder.

## Creating a Search Folder Programmatically

To create a search folder programmatically, you call the *AdvancedSearch* method of the *Application* object and pass the *Scope*, *Filter*, *SearchSubFolders*, and *Tag* parameters. The *AdvancedSearch* method returns a *Search* object. Once you have obtained a *Search* object, you can call the *Save* method on the *Search* object to create a search folder that is visible in the Outlook user interface, or you can examine the contents of the search programmatically without saving the search folder. The *GetTable* method of the *Search* object allows you to enumerate items in the *Search* object in a performant manner. Table 10-4 lists the parameters for the *AdvancedSearch* method.

**Table 10-4 Parameters for the *AdvancedSearch* Method**

| Name | Required? | Data type | Description |
|---|---|---|---|
| Scope | Required | String | The scope of the search; for example, the folder path of a folder. It is recommended that the folder path be enclosed within single quotes. Otherwise, the search might not return correct results if the folder path contains special characters, including Unicode characters. To specify multiple folder paths, enclose each folder path in single quotes and separate the single-quoted folder paths with a comma. |

**Table 10-4 Parameters for the *AdvancedSearch* Method**

| Name | Required? | Data type | Description |
|---|---|---|---|
| Filter | Optional | String | The DASL search filter that defines the parameters of the search. Do not prefix the DASL filter with the *@SQL=* prefix. |
| SearchSubFolders | Optional | Boolean | Determines if the search will include any of the folder's subfolders. If *SearchSubFolders* is *true* and multiple folders are specified by *scope*, then the subfolders of all folders specified in *scope* are searched. |
| Tag | Optional | String | The name given as an identifier for the search. |

The following extensive code sample provides an end-to-end illustration of how to create a search folder programmatically. The code creates a search folder that contains all items in the Inbox and RSS Subscriptions folders and their subfolders that contain items with "Office" in the subject. The search folder created by the sample code is shown in Figure 10-6.

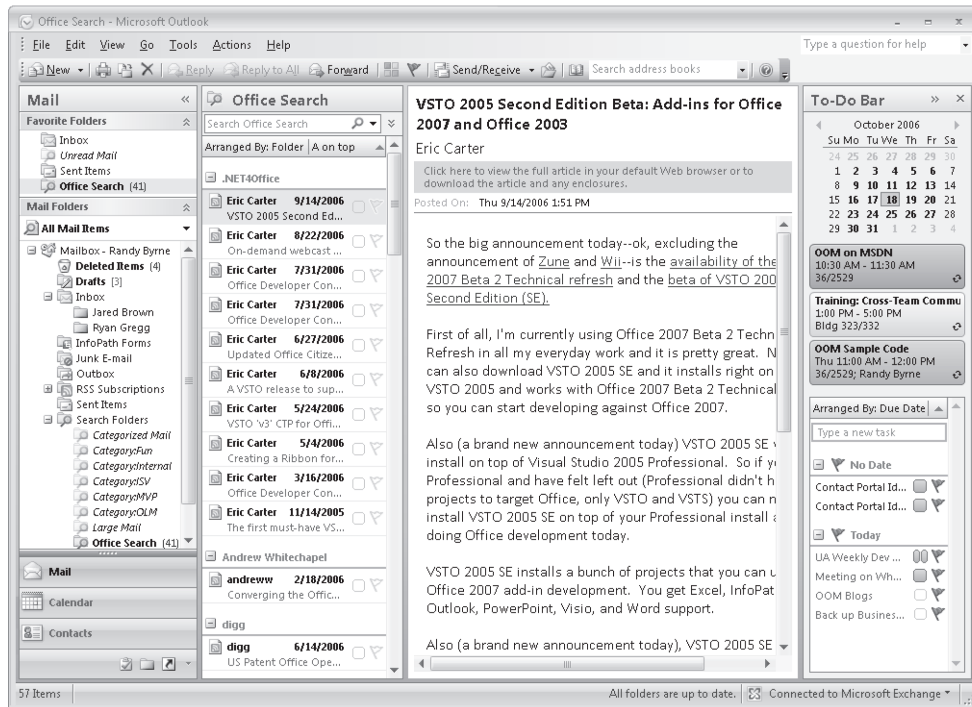


**Figure 10-6** Create the Office Search search folder programmatically.

The sample assumes that you are creating a search folder using an Outlook add-in. The *InitializeAddin* procedure is called by the add-in's *OnConnection* procedure.

```
private void InitializeAddin()
{
```

```
    Application.AdvancedSearchComplete += new
        Outlook.ApplicationEvents_11_AdvancedSearchCompleteEventHandler(
        Application_AdvancedSearchComplete);
    CreateOfficeSearch();
}

private void CreateOfficeSearch()
{
    // Construct search filter
    // Only use ci_ keywords if Instant Search is enabled
    string filter;
    if (Application.Session.DefaultStore.IsInstantSearchEnabled)
    {
        filter = "urn:schemas:httpmail:subject"
            + " ci_phrasematch 'Office'";
    }
    else
    {
        filter = "urn:schemas:httpmail:subject"
            + " like '%Office%'";
    }
    // Construct search scope
    StringBuilder sb = new StringBuilder();
    sb.Append("'");
    sb.Append(Application.Session.GetDefaultFolder(
        Outlook.OlDefaultFolders.olFolderInbox).FolderPath);
    sb.Append("'");
    sb.Append(",");
    sb.Append("'");
    sb.Append(Application.Session.GetDefaultFolder(
        Outlook.OlDefaultFolders.olFolderRssFeeds).FolderPath);
    sb.Append("'");
    string scope = sb.ToString();
    // Call AdvancedSearch method
    Outlook.Search search =
        Application.AdvancedSearch(
        scope, filter, true, "My Office Search");
    // To save the search as a search folder,
    // you can call Search.Save()
    search.Save("Office Search");
    // Add the search folder to favorites
    Outlook.Folder folder =
        Application.Session.DefaultStore.GetSearchFolders()
        ["Office Search"] as Outlook.Folder;
    Outlook.NavigationPane pane =
        Application.ActiveExplorer().NavigationPane;
    Outlook.MailModule mailModule =
        pane.Modules.GetNavigationModule(
        Outlook.OlNavigationModuleType.olModuleMail)
        as Outlook.MailModule;
    Outlook.NavigationGroup mailGroup =
        mailModule.NavigationGroups.GetDefaultNavigationGroup(
        Outlook.OlGroupType.olFavoriteFoldersGroup);
    mailGroup.NavigationFolders.Add(folder);
}
```

Each bullet in the following list discusses an important aspect of the sample code just shown:

- The *InitializeAddin* procedure creates an event handler for the *AdvancedSearchComplete* event on the *Outlook.Application* object and calls the *CreateOfficeSearch* procedure. Because *AdvancedSearch* returns results asynchronously, you need to create an event handler to determine when the search has completed.
- *CreateOfficeSearch* creates instance variables named *filter* and *scope*, and then passes those arguments to the *AdvancedSearch* method of the *Application* object. If Instant Search is enabled and *DefaultStore.IsInstantSearchEnabled* is *true*, then *filter* contains the *ci_phrasematch* keyword to create a phrase match search for "Office" in the item subject. If Instant Search is not enabled and *DefaultStore.IsInstantSearchEnabled* is *false*, then *filter* contains the *like* keyword to create a substring match search for "Office" in the item subject. Note that the *filter* does not impose an additional restriction for message class so that all item types (including meeting requests in the Inbox that contain "Office" in the subject) will be returned by the search. If you want to restrict by message class, you should add additional conditions to the criteria. The *scope* string specifies multiple folders for the search, namely the Inbox and RSS Subscriptions folders.
- *CreateOfficeSearch* calls the *AdvancedSearch* method of the *Application* object to return a *Search* object named *search*. The optional *SearchSubfolders* argument is *true* so that subfolders of the target folders will be searched. Also the *Tag* argument is specified so that the *Tag* property of the *Search* object will have the value My Office Search.
- *CreateOfficeSearch* saves the *Search* object named *search* returned by *AdvancedSearch*. The *Save* method is called on the *search* instance variable to persist the search as a search folder. The name of the search folder is Office Search. Although the code does not illustrate this precaution, you might want to check the *Folders* collection returned by *DefaultStore.GetSearchFolders()* to ensure that a search folder with same name does not already exist.
- Once the search folder has been saved, you can find the search folder in the *Folders* collection returned by *DefaultStore.GetSearchFolders()*. In this case, the code returns a *Folder* object that represents the newly created search folder.
- Now that you have an instance variable representing the search folder, you can use *NavigationPane* and related objects to add the newly created search folder to the user's favorite folders.
- Finally, the *AdvancedSearchComplete* method will fire when the search is complete. In the *Application_AdvancedSearchComplete* event procedure, the code checks that the *Search* object passed to the event is the search named My Office Search. You then use the *GetTable* method on the *Search* object and write the subject for every row in the table to the trace listeners in the *Listeners* collection.

# Outlook Views

Outlook 2007 allows you to create customizable views that allow you to better sort, group, and ultimately view data of all different types within the View Pane of Explorer. You can also customize built-in views programmatically. There are a variety of different view types that provide the flexibility needed to organize your solution's data. For example, Microsoft Business Contact Manager uses the custom view shown in Figure 10-7 to organize and present solution data in the view named By Campaign Type in the Marketing Campaigns folder.

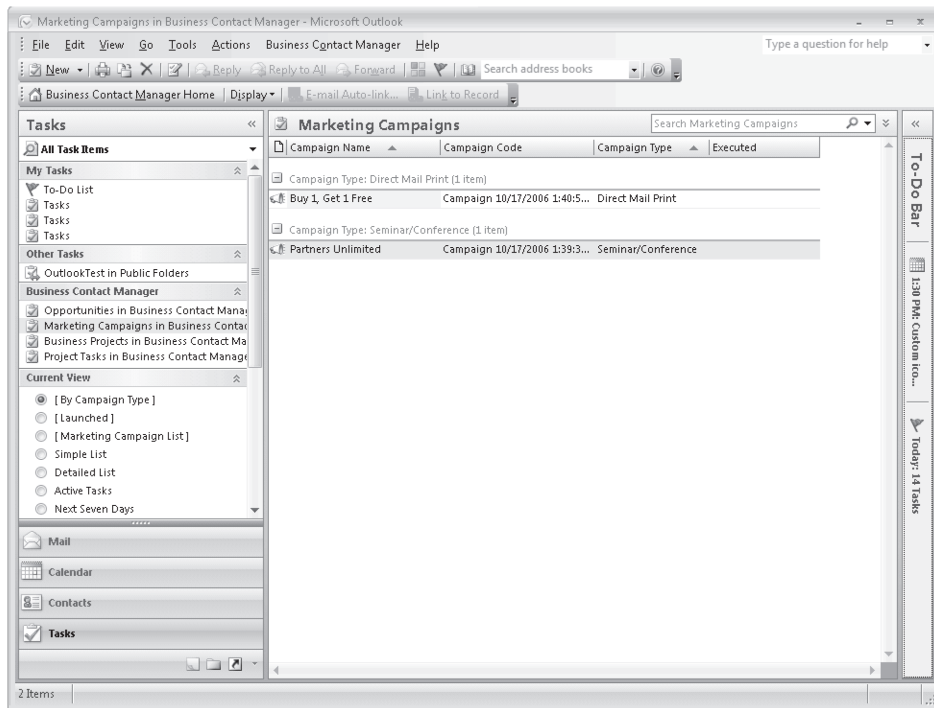


**Figure 10-7** Custom By Campaign Type view in the Marketing Campaigns folder.

## Objects That Derive from the *View* Object

Outlook 2007 supports the following objects that represent Outlook views. Table 10-5 lists new Outlook 2007 view objects that derive from the *View* object. For a complete listing of all the properties and methods of these view objects, see the Outlook Developer's Reference.

**Table 10-5 Outlook 2007 *View* Objects**

| Object name | Description |
|---|---|
| BusinessCardView | This object allows you to view data as a series of Electronic Business Card images. |
| CalendarView | This object allows you to view data in a calendar format. |

**Table 10-5 Outlook 2007 *View* Objects**

| Object name | Description |
|---|---|
| CardView | This object allows you to view data in a series of cards. |
| IconView | This object allows you to view data as icons, similar to a Windows folder or Explorer. |
| TableView | This object allows you to view data in a simple, field-based table. |
| TimelineView | This object allows you to view data in a customizable linear time line. |

Although you can use the *View* object to interact with the properties and methods common to all views, you must cast the *View* object to one of the derived view objects, such as the *CardView* object, to access certain properties, such as the *HeadingsFont* property of the *CardView* object. Use the *ViewType* property of the *View* object to determine which type of view is represented by that object. For example, the following code sample obtains the *CurrentView* object for the Inbox. If the *CurrentView* represents a *TableView* object, then the code creates an instance of the *TableView* and sets the *AllowInCellEditing* property to *true*. The code then calls the *Apply* method to reflect the change to the view in the Outlook user interface.

```
private void DemoAllowInCellEditingForView()
{
    Outlook.View view =
        Application.Session.GetDefaultFolder(
        Outlook.OlDefaultFolders.olFolderInbox).CurrentView;
    if (view.ViewType == Outlook.OlViewType.olTableView)
    {
        Outlook.TableView tableView = (Outlook.TableView)view;
        tableView.AllowInCellEditing = true;
        tableView.Apply();
    }
}
```

## Adding or Removing a View Programmatically

You can define a new view by using the *Add* method of the *Views* collection for a *Folder* object. Visibility for the view can be set either at the time of creation, by specifying an *OlViewSaveOption* constant in the *SaveOption* parameter of the *Add* method, or any time after the view is created, by specifying an *OlViewSaveOption* constant for the *SaveOption* property of the *View* object. Adding a new view raises the *ViewAdd* event of the *Views* collection. For example, the following code sample adds a new view named Meeting Requests to the user's Inbox. The DASL string supplied for the *Filter* property of the *View* object causes the view to display only items that contain "IPM.Schedule" in the message class for the item.

```
private void CreateMeetingRequestsView()
{
    const string PR_MESSAGE_CLASS =
        "http://schemas.microsoft.com/mapi/proptag/0x001A001E";
    Outlook.Views views =
        Application.Session.GetDefaultFolder(
```

```
        Outlook.OlDefaultFolders.olFolderInbox).Views;
    Outlook.TableView tableView = (Outlook.TableView)
        views.Add("Meeting Requests",
        Outlook.OlViewType.olTableView,
        Outlook.OlViewSaveOption.olViewSaveOptionThisFolderEveryone);
    tableView.Filter = "\"" + PR_MESSAGE_CLASS + "\"" +
        " like 'IPM.Schedule%'";
    tableView.Save();
    tableView.Apply();
}
```

If you need to remove a view from a folder, use the *Remove* method of the *Views* collection to remove an existing custom view. If you attempt to remove a built-in view, Outlook will raise an error. Removing a view raises the *ViewRemove* event of the *Views* collection.

Once a view is defined, you can customize the view programmatically by casting the *View* object to one of the derived view objects and performing whatever changes are needed. Use the *Save* method of the derived view object or the *View* object to save any changes to the view.

You can apply the view, once defined and customized, to the current *Explorer* object by using the *Apply* method of the derived view object or the *View* object. Applying a view raises the *ViewSwitch* event of the *Explorer* object.

# Customizing Your View

There are a variety of methods for customizing a built-in or custom view. In previous versions of Outlook, developers used the *XML* property of the *View* object to customize a view. In Outlook 2007, you can use the first-class properties of the derived *View* object to customize the view. Although the *XML* property of the *View* object is still available, you can achieve more consistent and easier results by using new view objects such as *ViewField*, *OrderField*, *ColumnFormat*, and *AutoFormatRule*.

## Specifying Fields in a View

You can specify which Outlook item properties are displayed in a view by adding one or more properties to the *ViewFields* collection of any of the following objects:

- *CardView*
- *TableView*

*BusinessCardView*, *CalendarView*, *IconView*, and *TimelineView* objects use other methods of determining which Outlook item properties are displayed within the view. The fields displayed for the *BusinessCardView* object, for example, are determined by the Electronic Business Card (EBC) layout associated with each displayed Outlook item.

The *ViewFields* collection for those views can be retrieved by accessing the *ViewFields* property of the appropriate *View* object. The *Add* method of the *ViewFields* collection is used to create a *ViewField* object that represents the Outlook item property to be displayed in the view.

**Note** To add built-in fields to the *ViewFields* collection, the property must exist in the Outlook field registry; otherwise Outlook will raise an error when you call the *Add* method. Use the Field Chooser to determine if the field exists in the Outlook field registry. To add custom fields to the *ViewFields* collection, the custom property must exist in the *UserDefinedProperties* collection of the parent *Folder* object; otherwise Outlook will raise an error when you call the *Add* method.

A *ViewField* object not only identifies an Outlook item property to display within the view, but also describes how the values for that property should be displayed. You can change how individual column properties are displayed in a view by modifying the *ColumnFormat* property of the *ViewField* object.

The following code sample adds the Start and End fields to the Meeting Requests view. It also changes the label for the From field to *Organized By*.

```
private void ModifyMeetingRequestsView()
{
    Outlook.TableView tableView = null;
    Outlook.ViewField startField = null;
    Outlook.ViewField endField = null;
    Outlook.ViewField fromField = null;
    try
    {
        tableView =
            Application.Session.GetDefaultFolder(
            Outlook.OlDefaultFolders.olFolderInbox)
            .Views["Meeting Requests"] as Outlook.TableView;
    }
    catch { }
    if (tableView != null)
    {
        try
        {
            startField = tableView.ViewFields["Start"];
        }
        catch{}
        if (startField == null)
        {
            startField = tableView.ViewFields.Add("Start");
        }
        try
        {
            endField = tableView.ViewFields["End"];
        }
        catch{}
        if (endField == null)
        {
```

```
                endField = tableView.ViewFields.Add("End");
            }
            try
            {
                fromField = tableView.ViewFields["From"];
            }
            catch{}
            if (fromField != null)
            {
                fromField.ColumnFormat.Label = "Organized By";
            }
            try
            {
                tableView.Save();
            }
            catch (Exception ex)
            {
                Debug.WriteLine(ex.Message);
            }
        }
    }
```

## Filtering Items in the *View* Object

Outlook items can be filtered in any view derived from the *View* object by specifying a valid DASL filter expression in the *Filter* property of the *View* object. Do not prefix the DASL string for the filter expression with *@SQL=* as you must for the *Restrict* method on the *Table* or *Items* objects. For more information about creating a DASL filter expression to filter Outlook items, see Chapter 11.

**Warning** Do not use *ci_phrasematch* and *ci_startswith* keywords in the filter expression for a view. The performance of the view will not be optimized if you use these keywords. For a view filter, use the = or *like* operators to construct your filter expression.

## Sorting Items in a View

Items in a view can be sorted by adding one or more Outlook item properties to the *OrderFields* collection of any of the following objects:

- BusinessCardView
- CardView
- IconView
- TableView

Outlook items in a *CalendarView* or *TimelineView* object are displayed in chronological order, depending on the values of the Outlook item properties specified for the *StartField* and *EndField* properties of the view.

The *OrderFields* collection for those views can be accessed with the *SortFields* property of the appropriate view object. The *Add* method of the *OrderFields* collection is used to create an *OrderField* object that represents the Outlook item property to be sorted.

### Specifying Properties for Sorting

You can add either built-in or custom Outlook item properties to the *OrderFields* collection. The order in which the properties are included in the *OrderFields* collection determines the order in which the properties are sorted, whereas the *IsDescending* property of the *OrderField* object, which represents an Outlook item property, determines whether the values of that property are sorted in ascending or descending order.

### Specifying Built-In Properties for Sorting

The following guidelines should be used when specifying built-in Outlook item properties:

- Built-in properties can be specified either by property name (for example, Subject) or by namespace (for example, *http://schemas.microsoft.com/mapi/proptag/0x0037001E*).
- Property names are not case-sensitive and cannot include spaces.

Namespace identifiers are case-sensitive, must follow URL encoding rules, and cannot be enclosed in square brackets ([]). For more information about property namespace identifiers, see Chapter 17, "Using the *PropertyAccessor* Object."

### Specifying Custom Properties for Sorting

The following guidelines should be used when specifying custom properties:

- The custom property must be available in the *UserDefinedProperties* collection for the parent *Folder* object.
- Custom properties should be specified by property name (for example, [Shoe Size]).
- Custom property names are not case-sensitive, can include spaces, and should be enclosed in square brackets ([]) if they contain spaces.

## The *AutoFormatRules* Collection

The new *AutoFormatRules* collection lets you add an *AutoFormatRule* object that represents a formatting rule used by a *View* object to determine how to format Outlook items displayed within that view.

Use the *Add* method or the *Insert* method of the *AutoFormatRules* collection to create a new formatting rule for the following objects:

- *CardView*
- *TableView*

For views that support automatic formatting, Outlook provides a set of built-in formatting rules that can be disabled but cannot be removed or reordered. Use the *Standard* property of the *AutoFormatRule* object to determine whether a formatting rule is built-in or custom. You cannot modify a built-in formatting rule. You can add or remove a custom formatting rule subject to the limitation that calling the *Save* or *Apply* methods will not persist *AutoFormatRule.Filter* in the *View* object. If you want to add an *AutoFormatRule* object to your solution, you need to add or remove the formatting rule dynamically.

The following *CreateAutoFormatRule* procedure creates a custom formatting rule named Canceled for the Meeting Requests view discussed earlier in this chapter. If the meeting item is a meeting cancellation, a red font is used to display the item in the view. To remove the formatting rule when the user navigates away from the folder or Outlook shuts down, the *RemoveAutoFormatRule* procedure deletes the Canceled formatting rule. The code sample assumes that you've created a class-level instance variable named *m_Explorer* and lists all the events necessary to make the dynamic formatting rule work correctly. For additional information on handling Outlook events, see Chapter 8, "Responding to Events."

```
private void InitializeAddin()
{
    m_Explorer = Application.ActiveExplorer();
    m_Explorer.BeforeViewSwitch += new
        Outlook.ExplorerEvents_10_BeforeViewSwitchEventHandler(
        m_Explorer_BeforeViewSwitch);
    m_Explorer.ViewSwitch += new
        Outlook.ExplorerEvents_10_ViewSwitchEventHandler(
        m_Explorer_ViewSwitch);
    Outlook.ExplorerEvents_Event explorerEvents =
        (Outlook.ExplorerEvents_Event)m_Explorer;
    explorerEvents.Close += new
        Outlook.ExplorerEvents_CloseEventHandler(m_Explorer_Close);
    m_Explorer.FolderSwitch += new
        Outlook.ExplorerEvents_10_FolderSwitchEventHandler(
        m_Explorer_FolderSwitch);
    if (m_Explorer.CurrentFolder.CurrentView.Name
        == "Meeting Requests")
    {
        CreateAutoFormatRule();
    }
}

void m_Explorer_FolderSwitch()
{
```

```
    if (m_Explorer.CurrentFolder.CurrentView.Name
        == "Meeting Requests")
    {
        CreateAutoFormatRule();
    }
}

void m_Explorer_Close()
{
     RemoveAutoFormatRule();
}

void m_Explorer_ViewSwitch()
{
    if (m_Explorer.CurrentFolder.CurrentView.Name
        == "Meeting Requests")
    {
        CreateAutoFormatRule();
    }
}

void  m_Explorer_BeforeViewSwitch(object NewView, ref bool Cancel)
{
    if (m_Explorer.CurrentFolder.CurrentView.Name
        == "Meeting Requests")
    {
        RemoveAutoFormatRule();
    }
}

private void CreateAutoFormatRule()
{
    Outlook.TableView tableView = null;
    Outlook.AutoFormatRule autoFormat = null;
    const string PR_MESSAGE_CLASS =
        "http://schemas.microsoft.com/mapi/proptag/0x001A001E";
    Outlook.Folder inbox = Application.Session.GetDefaultFolder(
        Outlook.OlDefaultFolders.olFolderInbox) as Outlook.Folder;
    Outlook.Folder currentFolder =
        Application.ActiveExplorer().CurrentFolder
        as Outlook.Folder;
    if (Application.Session.CompareEntryIDs(currentFolder.EntryID,
        inbox.EntryID))
    {
        try
        {
            tableView =
                inbox.Views["Meeting Requests"] as Outlook.TableView;
        }
        catch{ }
        if (tableView != null)
        {
            try
            {
```

```
                autoFormat =
                    tableView.AutoFormatRules["Canceled"];
            }
            catch{ }
            if (autoFormat == null)
            {
                autoFormat =
                    tableView.AutoFormatRules.Add("Canceled");
                autoFormat.Filter = "\"" + PR_MESSAGE_CLASS +
                    "\"" + " like '%Canceled%'";
                autoFormat.Font.Color = Outlook.OlColor.olColorRed;
                autoFormat.Enabled = true;
                // Save the view
                tableView.Save();
            }
        }
    }
}

private void RemoveAutoFormatRule()
{
    Outlook.TableView tableView = null;
    Outlook.AutoFormatRule autoFormat = null;
    Outlook.Folder inbox = Application.Session.GetDefaultFolder(
        Outlook.OlDefaultFolders.olFolderInbox) as Outlook.Folder;
    Outlook.Folder currentFolder =
        Application.ActiveExplorer().CurrentFolder
        as Outlook.Folder;
    if (Application.Session.CompareEntryIDs(currentFolder.EntryID,
        inbox.EntryID))
    {
        try
        {
            tableView =
                inbox.Views["Meeting Requests"] as Outlook.TableView;
        }
        catch { }
        if (tableView != null)
        {
            try
            {
                autoFormat =
                    tableView.AutoFormatRules["Canceled"];
            }
            catch { }
            if (autoFormat != null)
            {
                tableView.AutoFormatRules.Remove("Canceled");
                tableView.Save();
            }
        }
    }
}
```

# Summary

Outlook 2007 provides several features to help organize user or solution data. This chapter shows you how to leverage these features programmatically. You can use category colors, task flagging, rules, search folders, and views to organize or present data to the user. You learned how you can take advantage of these features in your solution and tailor them to your specific scenario.