MCAD/MCSD

Training Kit

Exam 70-310 and 70-320

DEVELOPING XML Web Services and Server Components WITH MICROSOFT® VISUAL BASIC .NET AND MICROSOFT® VISUAL C# .NET

Microsoft Press

MCAD/MCSD

Training Kit

Exam 70-306 and 70-316

DEVELOPING WINDOWS®-BASED APPLICATIONS WITH MICROSOFT® VISUAL BASIC .NET AND MICROSOFT® VISUAL C# .NET

Microsoft Press

Учебный
курс
MCAD/MCSD

**Сертификационный
экзамен 70-306 и 70-316**

Разработка Windows-приложений на Microsoft* Visual Basic .NET и Microsoft* Visual C# .NET

***Официальное** пособие Microsoft
для самостоятельной подготовки*

Москва 2003

РУССКАЯ РЕДАКЦИЯ

УДК 004
ББК 32.973.26–018.2
М59

Microsoft Corporation

М59 Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET. Учебный курс MCAD/MCSD/Пер. с англ. — М.: Издательско-торговый дом «Русская Редакция», 2003. — 512 стр.: ил.

ISBN 5–7502–0222–4

Этот учебный курс посвящен разработке Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET. В книге изложены основы объектно-ориентированного программирования для Windows, методы создания пользовательского интерфейса и работы с базами данных; значительное внимание уделяется вопросам защиты, тестирования и развертывания Windows-приложений, а также их глобализации и созданию справочной системы. Освоив теоретические материалы и выполнив практические задания курса, вы получите знания и навыки, необходимые разработчику Windows-приложений, использующих современную платформу Microsoft .NET.

Книга адресована всем тем, кто хочет научиться создавать Windows-приложения в среде Visual Studio .NET на языках Visual Basic .NET и C#. Настоящий учебный курс также поможет вам самостоятельно подготовиться к сдаче экзаменов по программе сертификации Microsoft (Microsoft Certified Application Developer и Microsoft Certified Solution Developer) № 70-306: Developing and Implementing Windows-based Applications with Microsoft Visual Basic .NET and Microsoft Visual Studio .NET и 70-316: Developing and Implementing Windows-based Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET.

Книга состоит из 10 глав, приложения и предметного указателя, содержит множество иллюстраций и примеров программного кода. На прилагаемом компакт-диске находятся словарь терминов, демонстрационные файлы, выполненные практические задания и др. справочные материалы.

УДК 004
ББК 32.973.26-018.2

Подготовлено к изданию по лицензионному договору с Microsoft Corporation. Редмонд, Вашингтон, США.

Active Directory, ActiveX, IntelliSense, JScript, Microsoft, Microsoft Press, Visual Basic, Visual C++, Visual C#, Visual Studio, Windows и Windows NT являются товарными знаками или охраняемыми товарными знаками корпорации Microsoft в США и/или других странах. Все другие товарные знаки являются собственностью соответствующих фирм.

Все названия компаний, организаций и продуктов, а также имена лиц, используемые в примерах, вымышлены и не имеют никакого отношения к реальным компаниям, организациям, продуктам и лицам.

© Оригинальное издание на английском языке, Microsoft Corporation, 2002
© Перевод на русский язык, Microsoft Corporation, 2003
© Оформление и подготовка к изданию, издательско-торговый лом «Русская Редакция», 2003

ISBN 0–7356–1533–0 (англ.)
ISBN 5–7502–0222–4

Содержание

Об этой книге . . . . . **XVI**

Глава 1 Введение в .NET Framework **1**
Занятие 1. Инфраструктура .NET Framework и общеязыковая исполняющая среда 2
Обзор .NET Framework 2
.NET Framework и языки программирования 2
Структура .NET-приложений 3
Компиляция и исполнение .NET-приложений 4
Резюме 5
Занятие 2. Библиотека базовых классов .NET 6
Типы значений и ссылочные типы 7
Применение типов .NET Framework в приложениях 8
Типы значений 8
Ссылочные типы 9
Применение переменных ссылочных типов и типов значений 10
Операторы Imports и using 12
Ссылки на внешние библиотеки 12
Резюме 13
Занятие 3. Классы и структуры 14
Члены 14
Создание классов 14
Создание структур 15
Добавление членов 15
Вложенные типы 16
Создание экземпляров пользовательских типов 16
Классы и структуры 18
Резюме 18
Занятие 4. Методы 19
Объявление методов 19
Вызов методов 20
Переменные методов 20
Параметры 21
Выходные параметры 23
Необязательные параметры 23
Конструкторы и деструкторы 24
Резюме 25
Занятие 5. Область видимости и уровни доступа 26
Модификаторы доступа к членам 26
Модификаторы доступа к типам 28
Модификаторы доступа для вложенных типов 29
Общие (статические) члены 29
Доступ к общим членам 30
Резюме 31
Занятие 6. Сбор мусора 32
Циклические ссылки 33
Резюме 34
Практикум 1. Применение классов и демонстрация сбора мусора 35

Упражнение 1. Создание класса Demo . . . 35
Упражнение 2. Демонстрация сбора мусора . . . 36
Закрепление материала . . . 38

Глава 2 Создание пользовательского интерфейса . . . 39
Занятие 1. Принципы разработки пользовательского интерфейса . . . 40
Формы, элементы управления и меню . . . 40
Принципы дизайна интерфейса . . . 41
Простота . . . 41
Композиция формы . . . 42
Согласованность . . . 42
Эстетика внешнего вида . . . 42
Резюме . . . 44
Занятие 2. Работа с формами . . . 45
Добавление формы к проекту . . . 45
Применение наследования с формами . . . 46
Назначение стартовой формы . . . 47
Выбор первоначального размещения формы . . . 48
Изменение внешнего вида формы . . . 49
Свойства BackColor, ForeColor и Text . . . 50
Свойства Font, Cursor и BackGroundImage . . . 50
Прозрачность . . . 50
Применение методов формы . . . 51
Методы Show и ShowDialog . . . 52
Метод Activate . . . 52
Метод Hide . . . 53
Метод Close . . . 53
Применение событий формы . . . 53
События жизненного цикла формы . . . 55
Событие Load . . . 55
События Activated и Deactivated . . . 56
Событие VisibleChanged . . . 56
Событие Closing . . . 56
Событие Closed . . . 56
Резюме . . . 57
Занятие 3. Применение элементов управления и компонентов . . . 58
Работа с элементами управления . . . 58
Установка порядка передачи фокуса при нажатии клавиши Tab . . . 60
Вложенные элементы управления и контейнеры . . . 61
Применение элементов управления Group Box и Panel . . . 61
Применение элемента управления TabControl . . . 61
Стыковка и фиксация элементов управления . . . 62
Применение свойства Anchor . . . 62
Применение свойства Dock . . . 64
Применение наборов элементов управления . . . 65
Добавление элементов управления на панель Toolbox . . . 66
Создание обработчиков событий элементов управления . . . 67
Использование провайдеров дополнительных свойств . . . 68
Резюме . . . 69
Занятие 4. Меню . . . 71
Конструирование меню во время разработки . . . 71
Применение компонента MainMenu . . . 71

Разделение элементов меню . . . 72
Доступ к меню с клавиатуры . . . 73
События элементов меню . . . 74
Создание контекстного меню . . . 74
Изменение меню во время выполнения . . . 75
Активация и деактивация команд меню . . . 75
Отображение элементов меню с флажками . . . 75
Отображение элементов меню с переключателями . . . 75
Как сделать элемент меню невидимым . . . 76
Клонирование меню . . . 76
Слияние меню во время выполнения . . . 77
Создание элементов меню во время выполнения . . . 77
Резюме . . . 78
Занятие 5. Проверка данных, вводимых пользователем . . . 79
Проверка на уровне поля . . . 79
Применение свойств элемента управления TextBox . . . 80
Применение событий для проверки на уровне поля . . . 80
Проверка вводимых символов . . . 82
Работа с фокусом ввода . . . 83
Проверка на уровне формы . . . 85
Обработка событий клавиатуры на уровне формы . . . 86
Оповещение пользователя об ошибках ввода . . . 86
Компонент ErrorProvider . . . 87
Отображение сообщений об ошибках . . . 87
Резюме . . . 89
Практикум 2. Приложение Virtual Doughnut Factory . . . 90
Упражнение 1. Создание пользовательского интерфейса . . . 90
Упражнение 2. Создание меню . . . 93
Упражнение 3. Создание обработчиков, проверяющих вводимые данные . . . 94
Закрепление материала . . . 97

Глава 3 Типы и члены . . . 99
Занятие 1. Применение типов данных . . . 100
Типы данных .NET Framework . . . 100
Целочисленные типы . . . 100
Типы значений с плавающей точкой . . . 101
Нечисловые типы . . . 102
Преобразование типов . . . 103
Неявное преобразование . . . 103
Явное преобразование . . . 104
Применение встроенной функциональности типов данных . . . 106
Упаковка . . . 106
Собственные методы типов данных . . . 106
Резюме . . . 109
Занятие 2. Константы, перечисления, массивы и наборы . . . 110
Константы и перечисления . . . 110
Применение констант . . . 110
Применение перечислений . . . 111
Массивы . . . 113
Объявление и инициализация массивов . . . 114
Многомерные массивы . . . 116
Наборы . . . 117

Класс ArrayList . . . 118
Другие типы наборов из класса Collections . . . 120
Перебор элементов массивов и наборов . . . 121
Резюме . . . 122
Занятие 3. Реализация свойств . . . 123
Реализация свойств . . . 123
Свойства, доступные только для чтения или только для записи . . . 125
Создание свойств, доступных только для чтения . . . 126
Свойства, доступные только для записи . . . 126
Параметризованные свойства . . . 127
Свойства по умолчанию и индексаторы . . . 128
Свойства-наборы . . . 129
Резюме . . . 131
Занятие 4. Реализация событий и делегатов . . . 132
Делегаты . . . 132
Объявление и генерация событий . . . 133
Реализация обработчиков событий . . . 135
Создание обработчиков событий в Visual Basic .NET . . . 135
Создание обработчиков событий на Visual C# . . . 136
Создание обработчика для группы событий . . . 136
События с несколькими обработчиками . . . 137
Отключение обработчиков во время выполнения . . . 137
Резюме . . . 138
Практикум 3. Добавление компонентов и реализация членов . . . 139
Упражнение 1. Создание компонента DoughnutMachine . . . 139
Упражнение 2. Подключение класса DoughnutMachine к пользовательскому интерфейсу . . . 146
Закрепление материала . . . 150

Глава 4 Объектно-ориентированное программирование и полиморфизм . . . 151
Занятие 1. Введение в объектно-ориентированное программирование . . . 152
Объекты, члены и абстрагирование . . . 152
Классы как шаблоны объектов . . . 152
Объекты и члены . . . 153
Объектные модели . . . 153
Инкапсуляция . . . 154
Полиморфизм . . . 154
Реализация полиморфизма через интерфейсы . . . 154
Реализация полиморфизма через наследование . . . 155
Резюме . . . 155
Занятие 2. Перегрузка членов . . . 156
Создание перегруженных методов . . . 157
Перегрузка операторов в Visual C# . . . 158
Резюме . . . 160
Занятие 3. Реализация полиморфизма через интерфейсы . . . 161
Определение интерфейсов . . . 161
Интерфейсы как средство достижения полиморфизма . . . 163
Реализация интерфейсов . . . 164
Реализация членов интерфейса на Visual C# . . . 165
Реализация членов интерфейса на Visual Basic .NET . . . 166
Резюме . . . 167

Занятие 4. Реализация полиморфизма через наследование . . . 168
Наследование . . . 168
Полиморфизм и производные классы . . . 168
Создание производных классов . . . 169
Создание запечатанных классов . . . 169
Наследуемые члены . . . 170
Переопределение членов, унаследованных от базового класса . . . 170
Замещение членов базового класса в Visual Basic .NET . . . 172
Сокрытие членов базового класса в Visual C# . . . 172
Обеспечение совместимости с замещенными или скрытыми членами . . . 173
Получение доступа к членам базового класса . . . 174
Защищенные члены . . . 175
Абстрактные классы и члены . . . 176
Создание абстрактных классов . . . 177
Создание абстрактных членов . . . 177
Наследование от абстрактного класса . . . 178
Резюме . . . 180
Практикум 4. Применение производных классов . . . 181
Упражнение 1. Создание класса строго типизированного набора на основе класса CollectionBase . . . 181
Упражнение 2. Создание дополнительных типов объектов Doughnut . . . 184
Упражнение 3. Реализация метода RemoveStale . . . 186
Закрепление материала . . . 191

Глава 5 Тестирование и отладка приложений . . . 193
Занятие 1. Использование отладочных инструментов . . . 194
Основные типы ошибок . . . 194
Синтаксические ошибки . . . 194
Ошибки периода выполнения . . . 195
Логические ошибки . . . 195
Пошаговое исполнение кода . . . 196
Step Into . . . 198
Step Over . . . 198
Step Out . . . 198
Run To Cursor . . . 198
Set Next Statement . . . 199
Установка точек прерывания . . . 199
Окно Breakpoints . . . 200
Удаление и отключение точек прерывания . . . 200
Применение отладочных инструментов . . . 200
Окно Output . . . 200
Окна Locals, Autos и Watch . . . 201
Резюме . . . 203
Занятие 2. Применение классов Debug и Trace . . . 205
Как работает трассировка . . . 205
Запись данных, генерируемых классами Trace и Debug . . . 206
Набор Listeners . . . 207
Запись трассировочных данных в текстовом формате . . . 208
Запись трассировочных данных в журнал событий . . . 209
Применение трассировочных переключателей . . . 210
Настройка трассировочных переключателей . . . 212
Резюме . . . 214

Занятие 3. Создание плана блочного тестирования . . . 215
План блочного тестирования . . . 215
Разработка контрольных примеров . . . 216
Составление тестовых данных . . . 217
Резюме . . . 219
Занятие 4. Обработка и генерация исключений . . . 220
Принципы обработки исключений . . . 220
Создание обработчиков исключений . . . 221
Генерация исключений . . . 225
Повторная генерация исключений . . . 225
Нестандартные исключения . . . 227
Резюме . . . 229
Практикум 5. Отладка приложения . . . 230
Упражнение 1. Поиск синтаксических ошибок . . . 230
Закрепление материала . . . 233

Глава 6 Доступ к данным с помощью ADO.NET . . . 235
Занятие 3. Обзор ADO.NET . . . 236
Доступ к отсоединенным данным . . . 236
Архитектура данных ADO.NET . . . 237
Объект DataSet . . . 237
Провайдер данных . . . 237
Объект Connection . . . 238
Объект Command . . . 238
Объект DataReader . . . 238
Объект DataAdapter . . . 239
Резюме . . . 239
Занятие 2. Доступ к данным . . . 240
Подключение к БД . . . 240
Команды для манипулирования данными . . . 243
Создание и настройка команд . . . 243
Исполнение команд . . . 244
Применение объекта DataReader . . . 247
Создание объекта DataReader . . . 248
Организация простого доступа к данным с помощью объекта DataReader . . . 248
Выборка столбца с помощью объекта DataReader . . . 249
Извлечение типизированных данных с помощью объекта DataReader . . . 250
Работа с несколькими результирующими наборами . . . 251
Формирование запросов SQL во время выполнения . . . 253
Создание и настройка объекта DataAdapter . . . 254
Создание DataAdapter с помощью окна Server Explorer . . . 254
Создание DataAdapter с помощью мастера Data Adapter Configuration Wizard . . . 255
Извлечение данных с помощью объектов DataAdapter . . . 256
Предварительный просмотр данных . . . 258
Типизированные объекты DataSet . . . 258
Резюме . . . 261
Занятие 3. Применение объектов DataSet и обновление данных . . . 262
Создание и заполнение DataSet без использования DataAdapter . . . 262
Доступ к «плоским» файлам . . . 264
Объект DataRelation . . . 266
Извлечение связанных записей . . . 267

Ограничения . . . 268
Ограничения Unique . . . 268
Ограничения Foreign Key . . . 269
Модификация и обновление данных . . . 270
Модификация данных . . . 270
Обновление базы данных . . . 272
Транзакции . . . 272
Обработка ошибок при обновлении базы данных . . . 274
Резюме . . . 277
Занятие 4. Связывание, просмотр и фильтрация данных . . . 278
Связывание данных . . . 278
Провайдеры данных . . . 278
Потребители данных . . . 279
Создание элемента управления, использующего простое связывание данных . . . 279
Связывание данных в период выполнения . . . 281
Управление потоком данных . . . 282
Сложное связывание данных . . . 285
Фильтрация и сортировка данных . . . 285
Создание объектов DataView . . . 286
Фильтрация и сортировка содержимого DataSet . . . 286
Модификация данных через объект DataView . . . 289
Класс DataViewManager . . . 289
Резюме . . . 290
Занятие 5. Применение XML в ADO.NET . . . 291
Извлечение данных XML из баз данных SQL Server 2000 . . . 291
Получение объекта XmlReader с помощью SqlCommand . . . 291
Применение XML с объектами DataSet . . . 292
Чтение XML-данных в объект DataSet . . . 292
Запись содержимого объекта DataSet в формате XML . . . 293
Применение класса XmlDataDocument . . . 294
Преобразования XSL . . . 296
Резюме . . . 297
Практикум 6. Подключение к базе данных . . . 298
Упражнение 1. Организация доступа к данным и работа с объектом DataReader . . . 298
Упражнение 2. Извлечение и обновление данных с помощью объектов DataAdapter и DataSet . . . 300
Упражнение 3. Создание типизированного объекта DataSet с помощью XML Designer . . . 302
Упражнение 4. Работа с мастером Data Form . . . 303
Закрепление материала . . . 306

Глава 7 Создание элементов управления с помощью .NET Framework . . . 307

Занятие 1. Применение GDI+ . . . 308
Пространства имен System.Drawing . . . 308
Объект Graphics . . . 309
Объекты Color, Brush и Pen . . . 312
Системные кисти, перья и цвета . . . 314
Рисование простых фигур . . . 315
Отображение текста . . . 316
Рисование сложных фигур . . . 317

Создание объекта GraphicsPath . . . 317
Резюме . . . 319
Занятие 2. Создание элементов управления . . . 320
Принципы создания элементов управления . . . 320
Наследование от существующих элементов управления . . . 321
Наследование от класса UserControl . . . 321
Наследование от класса Control . . . 322
Добавление членов к элементам управления . . . 322
Создание производных элементов управления . . . 323
Реализация новых возможностей в производных элементах управления . . . 323
Модификация внешнего вида существующих элементов управления . . . 324
Создание составных элементов управления . . . 325
Создание формируемых элементов управления . . . 327
Резюме . . . 329
Занятие 3. Решение типичных задач, связанных с разработкой элементов управления . . . 330
Добавление элементов управления на панель Toolbox . . . 330
Назначение значка элементу управления . . . 330
Отладка элементов управления . . . 331
Управление лицензированием элементов управления . . . 332
Загрузка элемента управления в Internet Explorer . . . 334
Резюме . . . 335
Практикум 7. Создание нестандартного элемента управления . . . 336
Упражнение 1. Создание элемента управления . . . 336
Упражнение 2. Тестирование элемента управления . . . 340
Закрепление материала . . . 342

Глава 8 Сложные случаи программирования с применением .NET Framework . . . 343
Занятие 1. Поддержка печати . . . 344
Компонент PrintDocument . . . 344
Создание объекта PrintDocument . . . 344
Как работает поддержка печати . . . 345
Событие PrintPage . . . 345
Вывод на печать . . . 346
Печать графики . . . 346
Печать текста . . . 348
Печать многострочного текста . . . 349
Работа с цветами . . . 351
Применение элемента управления PrintPreviewControl . . . 352
Элемент управления PrintPreviewDialog . . . 353
Настройка параметров печати . . . 353
PrintDialog . . . 354
PageSetupDiatog . . . 354
Конфигурирование PageSettings в период выполнения . . . 355
Резюме . . . 356
Занятие 2. Получение доступа к компонентам и их вызов . . . 357
Получение доступа к библиотекам типов .NET и COM . . . 357
Создание экземпляров элементов управления ActiveX . . . 358
Доступ к Web-сервисам . . . 359
Вызов функций Windows API . . . 363
Резюме . . . 364

Занятие 3. Реализация дополнительных возможностей доступа . . . 365
Реализация специальных возможностей на этапе проектирования . . . 365
Специальные возможности и логотип «Certified for Windows» . . . 366
Поддержка стандартных параметров . . . 366
Поддержка высококонтрастных цветовых схем . . . 366
Наличие документированных возможностей доступа с клавиатуры ко всем функциям программы . . . 367
Поддержка следования за фокусом ввода . . . 367
Использование различных средств отображения информации . . . 367
Специальные возможности и элементы управления Windows Forms . . . 367
Резюме . . . 368
Занятие 4. Создание справочной системы . . . 370
Работа с классом Help . . . 370
Применение компонента HelpProvider . . . 371
Резюме . . . 372
Занятие 5. Глобализация и локализация . . . 373
Определение глобализации и локализации . . . 373
Концепция культур . . . 373
Получение и установка текущей культуры . . . 374
Глобализация приложений . . . 375
Локализация приложений . . . 375
Получение и установка текущей культуры пользовательского интерфейса . . . 376
Создание локализованных форм . . . 376
Проверка информации, вводимой на разных языках . . . 378
Форматирование, зависимое от культуры . . . 378
Отображение текста справа налево . . . 379
Преобразование кодировки символов . . . 380
Резюме . . . 382
Практикум 8. Создание локализованной формы, поддерживающей печать . . . 383
Упражнение 1. Создание формы . . . 383
Упражнение 2. Локализация формы . . . 386
Упражнение 3. Создание поддержки печати . . . 389
Закрепление материала . . . 392

Глава 9 Конфигурирование и защита сборок . . . 393
Занятие 1. Сборки и ресурсы . . . 394
Сборки . . . 394
Создание сборок с библиотеками классов . . . 395
Ресурсы и ресурсные сборки . . . 396
Создание ресурсных файлов . . . 396
Внедрение ресурсов в сборки . . . 398
Создание ресурсных сборок . . . 398
Создание сопутствующих сборок . . . 399
Извлечение ресурсов во время выполнения . . . 399
Извлечение из сопутствующих сборок ресурсов, специфичных для культуры . . . 401
Разделяемые сборки . . . 401
Закрытые и разделяемые сборки . . . 402
Строгое имя . . . 402
Установка сборок в GAC . . . 403
Резюме . . . 404

Занятие 2. Конфигурирование и оптимизация приложений . . . 405
Создание конфигурационного файла . . . 405
Схема конфигурационного файла . . . 406
Динамические свойства . . . 407
Настройка динамических свойств посредством окна Properties . . . 407
Установка и извлечение динамических свойств вручную . . . 408
Оптимизация приложений . . . 410
Оптимизация на этапе разработки . . . 410
Стратегия оптимизации . . . 410
Измерение производительности . . . 411
Оптимизация средствами компилятора . . . 411
Резюме . . . 412
Занятие 3. Защита приложений . . . 413
Разрешения . . . 414
Интерфейс IPermission . . . 414
Модель защиты, основанная на ролях . . . 414
Объект Principal . . . 415
Императивная защита, основанная на ролях . . . 416
Декларативная защита, основанная на ролях . . . 418
Конфигурирование защиты по правам доступа к коду . . . 419
Разрешения на доступ к коду . . . 419
Создание разрешений на доступ к коду . . . 420
Члены класса CodeAccessPermission . . . 421
Императивная защита по правам доступа к коду . . . 421
Декларативная защита по правам доступа к коду . . . 424
Обработка исключений и императивная защита . . . 426
Резюме . . . 426
Практикум 9. Конфигурирование и защита приложения . . . 428
Упражнение 1. Создание конфигурационного файла . . . 428
Упражнение 2. Зашита приложения . . . 430
Закрепление материала . . . 432

Глава 10 Развертывание . . . 433
Занятие 1. Планирование развертывания приложения . . . 434
Развертывание приложений с помощью XCOPY . . . 434
Создание проекта установочной программы . . . 435
Мастер Setup Project . . . 435
Параметры компоновки установочной программы . . . 439
Output File Name . . . 439
Package Files . . . 439
CAB Size . . . 440
Bootstrapper . . . 440
Compression . . . 441
Подпись Authenticode . . . 441
Компоновка, распространение и развертывание проекта . . . 442
Компоновка . . . 442
Распространение на сменных носителях . . . 442
Распространение через сеть . . . 442
Распространение приложений через Интернет . . . 443
Развертывание проекта . . . 444
Резюме . . . 445

Занятие 2. Конфигурирование проекта установочной программы . . . 446
Проект установочной программы . . . 446
Регистрация компонентов во время установки . . . 447
Редакторы свойств программы установки . . . 448
Редактор файловой системы . . . 448
Редактор реестра . . . 450
Редактор типов файлов . . . 451
Редактор пользовательского интерфейса . . . 453
Редактор нестандартных действий . . . 455
Редактор условий установки . . . 457
Установка сборок в виде компилированных образов . . . 458
Проверка разрешений, запрашиваемых установленной сборкой . . . 459
Резюме . . . 459
Практикум]0. Создание установочной программы . . . 460
Упражнение 1. Создание проекта установочной программы . . . 460
Упражнение 2. Настройка установочной программы . . . 461
Упражнение 3. Установка приложения . . . 461
Закрепление материала . . . 462

Приложение Вопросы и ответы . . . 463

Предметный указатель . . . 473

06 этой книге

Мы рады представить вам учебный курс, посвященный разработке Windows-приложений на Visual Basic .NET и Visual C# .NET. Теоретический материал и практические задания из этой книги помогут вам получить знания и навыки, необходимые для разработки Windows-приложений на языках Visual Basic .NET и C#. Кроме того, вы сможете подготовиться к сдаче экзаменов 70-306 и 70-316 по программе сертификации Microsoft Certified Professional.

Примечание Подробно о программах сертификации специалистов Microsoft Certified Application Developer и Microsoft Certified Solution Developer рассказано в разделе «Программа сертификации специалистов Microsoft».

В разделе «Начало работы» этой главы указаны требования к оборудованию и программному обеспечению, а также параметры сетевой конфигурации, необходимые для выполнения некоторых практических заданий курса. Внимательно прочитайте его, прежде чем приступить к изучению материала.

Кому адресована эта книга

Данный курс предназначен разработчикам, которым приходится заниматься проектированием, реализацией и поддержкой Windows-приложений, а также всем, кто желает сдать следующие сертификационные экзамены:

- Developing and Implementing Windows-Based Applications with Microsoft Visual Basic .NET and Microsoft Visual Studio .NET (экзамен 70-306);
- Developing and Implementing Windows-Based Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET (экзамен 70-316).

Для изучения курса необходимо:

- уметь конструировать Windows-приложения на языках Visual Basic .NET или Visual *C#* в среде Visual Studio .NET;
- знать назначение и уметь применять основные элементы управления и меню в приложениях на Visual Basic .NET и Visual C#;
- понимать связь между элементами управления и событиями;
- знать основы языка SQL.

Содержимое компакт-диска

Компакт-диск учебного курса содержит ряд вспомогательных материалов, требуемых при изучении всего курса:

- **электронная книга.** На компакт-диске записана полная электронная версия материалов учебного курса;
- **результаты выполненных упражнений.** Каждая глава этого курса завершается упражнениями для закрепления навыков. На компакт-диске вы найдете результаты упражнения, с которыми можно будет сравнить собственные результаты. Эти материалы также вы можете использовать в качестве подсказки, если в ходе выполнения упражнения возникнут существенные трудности;
- необходимые файлы. Все файлы, необходимые для выполнения практических заданий, есть на компакт-диске; если потребуется, их можно скопировать оттуда;
- **примерные экзаменационные вопросы.** Они позволят вам получить представление о сертификационном экзамене, а также выяснить, насколько полно вы усвоили материал этого курса;
- **словарь терминов.**

Структура книги

Для повышения эффективности обучения главы этой книги разбиты на разделы:

- каждая глава начинается с раздела «В этой главе», содержащего краткий обзор обсуждаемых тем; следующий за ним раздел «Прежде всего» поможет подготовиться к изучению материала главы;
- главы делятся на занятия, посвященные отдельным темам. Каждое занятие включает теоретическую и практическую часть.

▶ **Для некоторых задач и упражнений практикумов даны пошаговые инструкции — они помечены так же, как этот абзац**

- в разделе «Резюме» подводятся краткие итоги занятия и формулируются основные выводы;
- упражнения в конце занятий (обозначаются значком на полях) позволят вам применить полученные навыки на практике или поэкспериментировать с приложением-примером, о котором шла речь в занятии. Где возможно, задания организованы так, чтобы к концу практикума получилось готовое приложение. Каждое упражнение посвящено отдельному этапу построения приложения;
- каждую главу завершает раздел «Закрепление материала». Отвечая на вопросы этого раздела, вы проверите, насколько твердо усвоили материал.

Приложение «Вопросы и ответы» содержит ответы на все вопросы книги.

Примечания

В этой книге используются различные виды примечаний:

- **Внимание!** — содержит сведения, необходимые для выполнения поставленной задачи;
- **Примечание** — содержит дополнительную информацию.

Обозначения

- **Вводимые вами символы или команды набраны** строчными буквами полужирного начертания.
- *Курсив* в операторах указывает, что в этом месте следует подставить собственные значения; элементы кода также напечатаны *курсивом.*

- Имена файлов и каталогов начинаются с Прописных Букв (за исключением имен, которые вы задаете сами). Кроме особо оговоренных случаев, для ввода имен файлов и каталогов в диалоговых окнах или в командной строке можно использовать строчные буквы.
- Расширения имен файлов набраны строчными буквами.
- Аббревиатуры напечатаны ПРОПИСНЫМИ БУКВАМИ.
- Примеры кода, текста, выводимого на экран и набираемого в командной строке, а также фрагменты инициализационных файлов выделены моноширинным шрифтом.
- В угловые скобки, < >, заключаются необязательные элементы. Например, наличие в синтаксисе команды элемента <*filename*>показывает, что здесь можно ввести имя файла. Сами скобки вводить не надо.
- Если в тексте соседствуют ключевые слова и операторы Visual Basic и Visual C#, первым приводится вариант для Visual Basic, а следом в скобках — вариант для Visual C#.
- Значками на полях помечены конкретные разделы.

Значок	Описание
	Файлы на компакт-диске. О назначении и расположении нужных файлов рассказывается в сопутствующем тексте
	Упражнения, выполнив которые вы приобретете необходимые навыки
	Вопросы для закрепления материала, которые позволят проверить, как вы усвоили материал данной главы. Ответы на них см. в приложении «Вопросы и ответы» в конце книги

Клавиатура

- Знак «+» между названиями клавиш означает, что их следует нажать одновременно. Например, выражение «Нажмите Alt+Tab» обозначает, что нужно нажать клавишу Tab, удерживая нажатой клавишу Alt.
- Запятая между названиями клавиш означает их последовательное нажатие. Например, выражение «Нажмите Alt, F, X» означает, что надо последовательно нажать и отпустить указанные клавиши. Если же указано «Нажмите Alt+W, L», то вам придется сначала нажать клавиши Alt и W вместе, потом отпустить их и нажать клавишу L.
- Команды меню можно выбирать с клавиатуры. Для этого нажмите клавишу Alt (чтобы активизировать меню), а затем последовательно — выделенные или подчеркнутые буквы в названиях нужных Вам разделов меню или команд. Кроме того, некоторым командам сопоставлены клавиатурные сокращения (они указаны в меню).
- Флажки и переключатели также можно устанавливать и снимать с клавиатуры. Для этого достаточно нажать AH, а затем клавишу, соответствующую подчеркнутой букве в названии флажка или переключателя. Кроме того, нажимая клавишу Tab, Вы можете сделать зону нужного параметра активной, а затем установить или снять выбранный флажок или переключатель, нажав клавишу «пробел».
- Работу с диалоговым окном всегда можно прервать, нажав клавишу ESC.

Обзор глав и приложений

Этот курс, предназначенный для самостоятельного изучения, включает материалы, которые помогут вам научиться создавать Web-приложения при помощи Visual Studio .NET. Курс рассчитан на последовательное изучение, но не исключена возможность работы лишь с интересующими вас главами (сориентироваться в структуре книги вам поможет следующий раздел). В этом случае обращайте внимание на раздел «Прежде всего» в начале каждой главы, где указаны предварительные требования и даны ссылки на главы, в которых можно найти материалы для подготовки к выполнению упражнений.

Ниже кратко описаны главы и приложения учебного курса.

- В главе «Об этой книге» собраны сведения о содержании учебника, а также о структурных единицах и условных обозначениях, принятых в нем. Внимательно прочитайте ее: это поможет вам эффективнее работать с материалами курса, а также выбрать интересующие вас темы.
- Глава 1 «Введение в .NET Framework» посвящена знакомству с инфраструктурой .NET Framework, общеязыковой исполняющей средой и библиотекой базовых классов .NET. Кроме того, в ней разъясняются принципы управления памятью с помощью сбора мусора, способы создания классов, структур и методов с различной областью действия.
- В главе 2 «Создание пользовательского интерфейса» обсуждаются компоненты и конструирование пользовательского интерфейса. В этой главе идет речь об элементах управления, демонстрируются способы работы с формами и событиями элементов управления, а также проверки данных, вводимых пользователями,
- В главе 3 «Типы и члены» подробно рассматривается создание и применение нестандартных типов. В частности, в этой главе рассказывается, как создают массивы и наборы, а также обсуждается реализация свойства и событий.
- Глава 4 «Объектно-ориентированное программирование и полиморфизм» посвящена концепциям объектно-ориентированного программирования. Здесь описано создание перегруженных типов и реализация полиморфизма с применением наследования и интерфейсов.
- В главе 5 «Тестирование и отладка приложений» описаны приемы отладки приложений с помощью инструментов, поддерживаемых Visual Studio .NET. Здесь рассказывается, как в реальном времени собирать сведения о работе приложения с помощью классов *Trace* и *Debug*, как создать план блочного тестирования приложения, а также описана генерация и обработка исключений.
- В главе 6 «Доступ к данным с помощью ADO.NET» в деталях рассматриваются приемы получения доступа к данным из различных источников при помощи ADO.NET. Разъясняются модели доступа к данным, ориентированые на использование отсоединенных и подсоединенных данных, а также связывание данных с элементами пользовательского интерфейса. В завершение разбираются приемы работы с данными в формате XML.
- В главе 7 «Создание элементов управления с помощью .NET Framework» речь идет о создании составных, производных и формируемых элементов управления, о рисовании графики с помощью функций GDI+ и приводится методика решения задач, часто возникающих во время создания элементов управления.
- В главе 8 «Сложные случаи программирования с применением .NET Framework» рассказано о решении усложненных задач программирования. В частности, рас-

сматривается создание локализованных приложений, организация поддержки печати, а также получение доступа к компонентам COM, вызов функций Windows API и Web-сервисов. Заключительные разделы этой главы посвящены созданию справочной системы и реализации дополнительных возможностей доступа.

- В главе 9 «Конфигурирование и защита сборок» рассматривается создание сборок и работа с ресурсами, извлечение из конфигурационных файлов значений для свойств различных объектов приложения, а также реализация системы безопасности, основанной на ролях, и применение защиты по правам доступа к коду.
- В главе 10 «Развертывание» описано, как создать и сконфигурировать проект установочной программы для приложения, а также разъясняются принципы планирования развертывания приложений.
- В приложении «Вопросы и ответы» приведены ответы на вопросы из разделов «Закрепление материала» всех глав учебного курса.

С чего начать

Данный курс предназначен для самостоятельного изучения, поэтому вы можете пропускать некоторые занятия, чтобы вернуться к ним позже. Чтобы определить, с чего начать изучение курса, обратитесь к этой таблице.

Если вы	Что делать
готовитесь к сдаче сертификационного экзамена 70-306 или 70-316	см. раздел «Начало работы», затем изучите книгу в удобном вам порядке
хотите изучить информацию по определенной теме экзамена	см. раздел «Материалы для подготовки к экзаменам»

Материалы для подготовки к экзаменам

В таблицах перечислены темы сертификационных экзаменов 70-306: *Developing and Implementing Windows-Based Applications with Microsoft Visual Basic .NET and Microsoft Visual Studio .NET* и 70-316: *Developing and Implementing Windows-Based Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET*, а также главы настоящего учебного курса, где обсуждаются соответствующие вопросы.

Примечание Конкретное содержание любого экзамена определяется компанией Microsoft и может быть изменено без предварительного уведомления.

Создание сервисов, предназначенных для пользователей

Тема	Где обсуждается Глава	Занятие
Создание Windows-форм с помощью дизайнера форм	2	2-1
Размещение элементов управления на Windows-формах	2	2-3
Отображение форм пользовательского интерфейса и реализация переходов между ними	2	2-2
Проверка информации, вводимой пользователем	2	2-5

(окончание)

Тема	Где обсуждается Глава	Занятие
Обработка ошибок, связанных с пользовательским интерфейсом	5	5-4
Создание встроенной справочной системы	8	8 4
Отображение содержимого источника данных	6	*6-4*
Вызов имеющегося кода из Windows-приложений	8	8-2
Создание и вызов экземпляров Web-сервисов и других компонентов	8	8 2
Глобализация приложений	8	8-5
Объявление, реализация и обработка событий	3	3 4
Организация поддержки печати	8	8-1
Реализация дополнительных возможностей доступа к приложению	8	8-3

Создание компонентов и .NET-сборок и управление ими

Тема	Где обсуждается Глава	Занятие
Создание и модификация .NET-сборок	9	9 1
Создание Windows-элементов управления	7	*7-2*

Работа с данными и манипулирование ими

Тема	Где обсуждается Глава	Занятие
Доступ к базам данных Microsoft SQL Server и манипулирование содержащейся в них информацией при помощи запросов и хранимых процедур	6	6-1, 2, 3
Доступ к хранилищам данных и манипулирование содержащейся в них информацией	6	6-1, 2, 3
Обработка ошибок, возникающих при работе с данными	6	6-3

Тестирование и отладка

Тема	Где обсуждается Глава	Занятие
Создание плана блочного тестирования	5	ɔ 3
Применение трассировки	5	5-2
Отладка, доработка и устранение дефектов кода	5	5-1

Развертывание Windows-приложений

Тема	Где обсуждается Глава	Занятие
Планирование развертывания Windows-приложения	10	1
Создание установочной программы, поддерживающей удаление приложения	10	1
Развертывание Windows-приложения	10	1
Добавление сборок в глобальный кэш сборок	9, 10	9-1, 10-2
Проверка соответствия разрешений, запрошенных устанавливаемым приложением, политике безопасности	10	10-2

Сопровождение и поддержка Windows-приложений

Тема	**Где обсуждается Глава**	**Занятие**
Оптимизация производительности Windows-приложений	5, 9	5-2, 9-2
Диагностика ошибок и устранение сбоев	5, 9	5-1, 2

Конфигурирование и зашита Windows-приложений

Тема	**Где обсуждается Глава**	**Занятие**
Конфигурирование Windows-приложений	9	9-2
Защита Windows-приложений	9	9-3
Настройка авторизации	9	9-3

Начало работы

Данный курс предназначен для самостоятельного изучения и содержит упражнения и практические рекомендации, которые помогут вам освоить разработку Windows-приложений на Visual Basic .NET и Visual C# .NET.

Аппаратное обеспечение

Компьютер должен соответствовать приведенной ниже минимальной конфигурации, а установленное на нем оборудование необходимо выбрать из списка совместимых устройств Microsoft Windows XP или Microsoft Windows 2000 Hardware Compatibility List:

- процессор Pentium II 450 МГц;
- 160 Мб памяти (рекомендуется 256 Мб);
- 12-скоростной привод CD-ROM или более быстрый;
- 3,5 Гб свободного пространства на жесткомдиске, в том числе 500 Мб в системном разделе;
- монитор SVGA (256 цветов, рекомендуется разрешение 800x600 или более высокое);
- мышь Microsoft или другое аналогичное устройство;

Программное обеспечение

Для выполнения упражнений вам потребуется следующее ПО:

- Microsoft Windows XP Professional Edition или Windows 2000;
- Microsoft Visual Studio .NET Professional Edition, Enterprise Developer Edition или Visual Studio .NET Enterprise Developer.[1]

Подготовка компьютера к выполнению упражнений

Прежде всего настройте компьютер согласно инструкциям поставщика.

Установка файлов, необходимых для занятий

Чтобы установить на жесткий диск файлы, необходимые для выполнения занятий, выполните следующие действия.

1. Вставьте прилагаемый компакт-диск в дисковод CD-ROM своего компьютера.

   Примечание Если на вашем компьютере отключена функция автозапуска, следуйте указаниям из файла Readme.txt на компакт-диске.

2. Щелкните в открывшемся меню ссылку Lab Folder и выберите папку с данными для интересующего вас упражнения. При необходимости скопируйте эти файлы в рабочий каталог на жестком диске.

Электронная версия книги

На прилагаемом компакт диске вы найдете электронную версию этой книги, которую можно просматривать через HTML-бразуер.

Чтобы установить электронную книгу, выполните следующие действия,

1. Вставьте прилагаемый компакт-диск в дисковод CD-ROM своего компьютера.

   Примечание Если на вашем компьютере отключена функция автозапуска, следуйте указаниям из файла Readme.txt на компакт-диске.

2. Щелкните в открывшемся меню ссылку Training Kit eBook и следуйте указаниям программы.

   Примечание Для работы с электронной книгой необходимо наличие прилагаемого компакт-диска в приводе CD-ROM вашего компьютера.

Вопросы пробного экзамена

Чтобы установить на жесткий диск вопросы пробного экзамена, выполните следующие действия.

1. Вставьте прилагаемый компакт-диск в дисковод CD-ROM своего компьютера.

   Примечание Если на вашем компьютере отключена функция автозапуска, следуйте указаниям из файла Readme.txt на компакт-диске.

2. Щелкните в открывшемся меню ссылку Sample Exam Questions и выберите нужный тест.

[1] Бесплатную **пробную** версию Visual Studio .NET **можно** загрузить с Web-сайта Microsoft по адресу *http://msdn.microsoft.com/vstudio. — Прим. ред.*

Программа сертификации специалистов Microsoft

Программа сертификации специалистов Microsoft (Microsoft Certified Professional, MCP) — отличная возможность подтвердить Ваше знание современных технологий и программных продуктов этой фирмы. Лидер отрасли в области сертификации, Microsoft разработала современные методы тестирования. Экзамены и программы сертификации подтвердят Вашу квалификацию разработчика или специалиста по реализации решений на основе технологий и программных продуктов Microsoft. Сертифицированные Microsoft профессионалы квалифицируются как эксперты и высоко ценятся на рынке труда.

Программа сертификации специалистов предлагает 7 типов сертификации по разнымспециальностям.

- *Сертифицированный разработчик приложений Microsoft (Microsoft Certified Application Developer, MCAD) для платформы Microsoft .NET* — способен разрабатывать и поддерживать приложения уровня департамента, а также серверные и клиентские компоненты служб для работы с данными, предназначенных для использования как в Web, так и в корпоративных сетях,
- *Сертифицированный разработчик программных решений Microsoft (Microsoft Certified Solution Developer, MCSD)* — проектирование и разработка решений для бизнеса с использованием средств разработки, платформ и технологий корпорации Microsoft, а также архитектуры Microsoft Windows;
- *Сертифицированный специалист Microsoft (Microsoft Certified Professional, MCP)* — предполагает доскональное знание по крайней мере одной ОС из семейства Windows или ключевой платформы Microsoft. Такой специалист обладает навыками внедрения продукта или технологии Microsoft как части бизнес-системы предприятия.
- *Сертифицированный системный инженер Microsoft (Microsoft Certified System Engineer, MCSE)* ~ предполагает умение эффективно анализировать потребности компаний, а также проектировать и реализовать инфраструктуры для бизнес-решений на базе платформы Windows 2000 и серверных продуктов корпорации Microsoft.
- *Сертифицированный системный администратор Microsoft (Microsoft Certified System Administrator, MCSA) на Windows 2000* — занимается вопросами реализации, управления и устранения неполадок в существующих системах на основе Windows 2000 и включая Windows .NET Server.
- *Сертифицированный администратор баз данных Microsoft (Microsoft Certified Database Administrator, MCDBA) на Microsoft SQL Server 2000* — разработка физической структуры, логических моделей данных, создание физических БД, создание служб доступа к данным с использованием T-SQL, управление и поддержка БД, настройка и управление системой защиты, мониторинг и оптимизация БД, а также установка и настройка Microsoft SQL Server.
- *Сертифицированный преподаватель Microsoft (Microsoft Certified Trainer, MCT)* ~ теоретическая и практическая подготовка для ведения соответствуюших курсов с использованием учебных материалов Microsoft Official Curriculum (MOC) в сертифицированных центрах технического обучения Microsoft (Microsoft Certified Technical Education Centers, CTECs).

Достоинства сертификации Microsoft

Программа сертификации Microsoft — один из самых строгих и полных тестов оценки знаний и навыков в области проектирования, разработки и сопровождения программного обеспечения. Сертификации удостаивается лишь тот, кто демонстрирует умение решать конкретные задачи, применяя продукты компании. Программа тестирования позволяет не только оценить квалификацию специалиста, но и служит ориентиром для всех, кто стремится достичь современного уровня знаний в этой области. Как и любой другой тест или экзамен, сертификация Microsoft является показателем определенного уровня знаний специалиста, что важно для работодателя и всей организации в целом.

Преимущества сертифицированного специалиста

Звание Microsoft Certified Professional дает вам:

- официальное признание ваших знаний и опыта работы с продуктами и технологиями Microsoft;
- подписку на Microsoft Developer Network. MCP получают скидку на годовую подписку на Microsoft Developer Network (см. *msdn.microsoft.com/subscriptions/*) в течение первого года действия сертификации (более точную информацию об условиях подписки вы найдете в MCP Welcome Kit);
- доступ к технической информации о продуктах Microsoft через защищенную область Web-узла MCP (см. *http://www.microsoft.com/traincert/mcp/mcpsecure.asp/*);
- доступ к эксклюзивным скидкам на продукты и услуги от определенных компаний. Сертифицированные специалисты могут узнать больше о предлагаемых скидках на защищенной области сайта MCP в разделе Other Benefits (см. *http://www.microsoft.com/traincert/mcp/mcpsecure.asp/*);
- право использовать логотип MCP, транскрипт, сертификат, карточку и значок MCP, чтобы проинформировать ваших клиентов и коллег об имеющемся статусе Microsoft Certified Professional. Электронные файлы логотипа MCP и ваш персональный транскрипт могут быть получены через защищенную облает], сайта MCP (*http://www.microsoft.com/traincert/mcp/mcpsecure.asp/*);
- приглашения на конференции, семинары и специальные мероприятия Microsoft, предназначенные для специалистов;
- свободный доступ к защищенной области Microsoft Certified Professional Magazine Online — журнала, посвященного вопросам карьеры и профессионального развития сертифицированных специалистов Microsoft, а также к разделам Web-сайта журнала со статьями и другой информацией, доступной только на сайте, и регулярно проводимым форумам с участием экспертов Microsoft и других компаний;
- скидку на членство в Professional Association for SQL Server (PASS, Профессиональная Ассоциация пользователей SQL Server) — единственной всемирной организации, управляемой пользователями SQL Server и функционирующей с одобрения Microsoft. Ее членам доступны уникальные возможности повышения уровня своих знаний и навыков. Обратитесь на защищенную область сайта MCP (*http://www.microsoft.com/traincert/mcp/mcpsecure.asp/*) для получения дополнительной информации о том, как воспользоваться данным преимуществом.

Кроме того, в зависимости от типа сертификации и страны, обладатели сертификата MCSA или MCDBA получают 50-процентную скидку на годовую подписку

на компакт-диски с материалами TechNet и TechNet Plus в течение первого года действия сертификации (более точную информацию об условиях подписки вы найдете в MCP Welcome Kit). Эти специалисты также могут получить бесплатный доступ к большинству материалов, содержащихся на компакт-дисках, через сайт TechNet (*http://www.microsoft.com/technet/*).

Обладатели сертификата MCDBA дополнительно получают бесплатную годовую подписку на журнал SQL Server Magazine, авторами которого являются ведущие специалисты отрасли. Этот журнал, содержащий массу полезной технической информации и практических советов, необходим каждому, кто работает с SQL Server.

Список преимуществ сертифицированных преподавателей Microsoft можно найти на сайте *http://www.microsoft.com/traincert/mcp/mct/benefits.asp*.

Выигрыш от сертификации Microsoft для работодателей и организаций

Сертификация позволяет организациям быстро окупить затраты на технологии Microsoft и извлечь максимум прибыли из этих технологий. Исследования показывают, что сертификация сотрудников по программам Microsoft:

- быстро окупается за счет стандартизации требований к обучению специалистов и методов оценки их квалификации;
- позволяет увеличить эффективность обслуживания клиентов, повысить производительность труда и снизить расходы на сопровождение ПО;
- обеспечивает надежные критерии для найма специалистов и их продвижения по службе;
- предоставляет методы оценки эффективности труда персонала;
- обеспечивает гибкие методы переподготовки сотрудников для обучения новым технологиям;
- позволяет оценить партнеров — сторонние фирмы.

Требования к соискателям

Требования к соискателям определяются специализацией, а также служебными функциями и задачами.

Соискатель сертификата Microsoft должен сдать экзамен, подтверждающий его глубокие знания в области программных продуктов Microsoft. Экзаменационные вопросы, подготовленные с участием ведущих специалистов компьютерной отрасли, отражают реалии применения программных продуктов Microsoft.

- На звание *Сертифицированного специалиста Microsoft* сдают экзамен по работе с одной из операционных систем. Кандидат может сдать дополнительные экзамены, которые подтвердят его право на работу с другими продуктами, инструментальными средствами или прикладными программами Microsoft.
- На звание *Сертифицированного системного инженера Microsoft* сдают семь экзаменов: пять ключевых и два по выбору.
- На звание *Сертифицированного системного администратора Microsoft* сдают четыре экзамена: три ключевых и один по выбору.
- На звание *Сертифицированного администратора баз данных Microsoft* сдают три ключевых экзамена и один по выбору.
- На звание *Сертифицированного разработчика программных решений на основе технологий Microsoft* сдают четыре экзамена по основам технологий ОС Microsoft Windows и один по выбору.

- На звание *Сертифицированного преподавателя Microsoft* надо подтвердить свою теоретическую и практическую подготовку для ведения соответствующих курсов в авторизованных учебных центрах Microsoft. Участие в программе требует соответствия требованиям, предъявляемым при ежегодном обновлении статуса сертифицированного преподавателя. Более подробные сведения о сертификации по этой программе можно получить на сайте *http://www.microsqft.com/traincert/mcp/mct/* или в местном отделении компании Microsoft.

Подготовка к экзаменам

Предусмотрено три режима подготовки: самостоятельная работа, интерактивный режим, а также занятия с инструктором в авторизованных центрах подготовки.

Самостоятельная подготовка

Самостоятельная подготовка — наиболее эффективный метод подготовки для инициативных соискателей. Издательство Microsoft Press предлагает весь спектр учебных пособий для подготовки к экзаменам по программе сертификации специалистов Microsoft. Учебные курсы для самостоятельного изучения, адресованные специалистам компьютерной отрасли, содержат теоретические и практические материалы, мультимедийные презентации, упражнения и необходимое ПО. Серия «Mastering» — это интерактивные обучающие компакт-диски для опытных разработчиков. Все эти пособия позволят Вам наилучшим образом подготовиться к сдаче сертификационных экзаменов.

Интерактивная подготовка

Интерактивная подготовка средствами Интернета — альтернатива занятиям в учебных центрах. Вы можете выбрать наиболее удобный распорядок занятий в виртуальном классе, где Вы научитесь работать с продуктами и технологиями Microsoft и подготовитесь к сдаче экзаменов. Интерактивное обучение предусмотрено для множества курсов Microsoft — от обычных официальных до специальных, доступных только в таком режиме. Интерактивные ресурсы доступны круглосуточно в авторизованных центрах подготовки.

Сертифицированные центры технического обучения

Сертифицированные центры технического обучения (Microsoft Certified Technical Education Center, CTEC) — самый простой способ пройти курс обучения пол руководством опытного инструктора и стать сертифицированным специалистом. Microsoft CTEC — всемирная сеть учебных центров, которые позволяют специалистам повысить свой технический потенциал под руководством сертифицированных инструкторов Microsoft.

Список центров CTEC в США и Канаде можно получить на Web сайте *http://www.microsoft.com/traincert/ctec/*.

Техническая поддержка

Мы постарались сделать все от нас зависящее, чтобы и учебный курс, и прилагаемый к нему компакт-диск не содержали ошибок. Издательство Microsoft Press публикует постоянно обновляемый список исправлений и дополнений к своим книгам по адресу *http://mspress.microsoft.com/support/*.

Если все же у вас возникнут вопросы или Вы захотите поделиться своими предложениями или комментариями, обращайтесь в издательство Microsoft Press по одному из указанных ниже адресов:

Электронная почта: TKINPUT@MICROSOFT.COM

Почтовый адрес: Microsoft Press
Attn: *MCAD/MCSD Training Kit: Developing Web Applications with Microsoft Visual Basic .NET and Visual C# .NET* Editor
One Microsoft Way
Redmond, WA 98052-6399

ГЛАВА 1

Введение в .NET Framework

Занятие 1. Инфраструктура .NET Framework и общеязыковая исполняющая среда 2

Занятие 2. Библиотека базовых классов .NET 6

Занятие 3. Классы и структуры 14

Занятие 4. Методы 19

Занятие 5. Область видимости и уровни доступа 26

Занятие 6. Сбор мусора 32

В этой главе

В этой главе обсуждаются инфраструктура .NET Framework и *общеязыковая исполняющая среда* (Common Language Runtime, **CLR), а** также основы синтаксиса объявления классов, структур и методов.

Прежде всего

Для освоения материалов этой главы не требуется предварительной подготовки,

Занятие 1. Инфраструктура .NET Framework и общеязыковая исполняющая среда

Инфраструктура .NET Framework — это интегрированная управляемая среда для создания и исполнения кода. На этом занятии вы познакомитесь с инфраструктурой .NET Framework, ее философией и принципами работы.

Изучив **материал** этого занятия, вы сможете:

- ✓ описать элементы .NET Framework;
- ✓ рассказать, из каких частей состоит сборка, и что содержится в каждой из них;
- ✓ объяснить, как компилируются и исполняются .NET-приложения.

Продолжительность занятия - около 20 минут.

Обзор .NET Framework

.NET Framework — это управляемая среда для разработки и исполнения приложений, обеспечивающая контроль типов. Эта среда управляет выполнением программы: она выделяет память под данные и команды, назначает разрешения программе или отказывает в их предоставлении, начинает исполнение приложения и управляет его ходом, а также отвечает за освобождение и повторное использование памяти, занятой ресурсами, более ненужными программе. .NET Framework состоит из двух основных компонентов: общеязыковой исполняющей среды (CLR) и библиотеки классов .NET Framework.

CLR можно рассматривать как среду, управляющую исполнением кода и предоставляющую ключевые функции, такие, как компиляция кода, выделение памяти, управление потоками и сбор мусора. Благодаря использованию *общей системы типов* (common type system, CTS) CLR выполняет строгую проверку типов, а защита по правам доступа к коду позволяет гарантировать исполнение кода в защищенном окружении.

Библиотека классов .NET Framework содержит набор полезных типов, разработанных специально для CLR и доступных для многократного использования. Типы, поддерживаемые .NET Framework, являются объектно-ориентированными, полностью расширяемыми и обеспечивают бесшовную интеграцию приложений с .NET Framework. Библиотека базовых классов .NET обсуждается на занятии 2.

.NET Framework и языки программирования

Конструкция .NET Framework обеспечивает межъязыковую совместимость. Проще говоря, компоненты, реализованные с применением .NET Framework, способны взаимодействовать друг с другом независимо от языка, на котором они написаны. Так, приложение на Visual Basic .NET может обращаться к DLL, написанной на C#, а та, в свою очередь, способна вызвать ресурсы, созданные на управляемом C++ или любом другом .NET-совместимом языке. Межъязыковая совместимость поддерживается и для наследования в ООП (объектно-ориентированном программировании), например, на основе C#-класса можно объявлять классы в программах на Visual Basic .NET и наоборот.

Столь высокий уровень межъязыковой совместимости стал возможным благодаря использованию общеязыковой исполняющей среды. Во время компиляции .NET-приложения, написанного на языке высокого уровня (таком, как Visual Basic .NET, C# или другом .NET-совместимом языке) генерируются команды языка *Microsoft Intermediate Language* (MSIL или просто IL). IL — это специальный низкоуровневый язык, который «понимает» CLR. Все DLL и исполняемые файлы .NET содержат команды этого языка, что позволяет им легко взаимодействовать. Стандарт под названием *общеязыковая спецификация* (Common Language Specification) определяет минимальный набор правил, которым должны следовать компиляторы для .NET-языков, чтобы откомпилированный ими код мог взаимодействовать с .NET Framework.

Использование CTS обеспечивает совместимость типов между .NET-компонентами. Поскольку перед развертыванием и исполнением .NET-приложений выполняется их компиляция в IL-код, все элементарные типы данных будут представлены типами, поддерживаемыми .NET. Так, типы *Integer* (Visual Basic) и *int* (Visual C#) в IL-коде представлены типом *System.Int32*. Применение в обоих языках общей системы типов, которые можно преобразовать друг в друга, позволяет компонентам обмениваться данными, избегая потерь времени на преобразование типов и «отлов» неизбежных при этом ошибок.

С Visual Studio .NET поставляются языки Visual Basic .NET, Visual C#, Visual C++ с управляемыми расширениями, а также язык сценариев JScript. Есть и другие языки, на которых можно писать управляемый код для .NET Framework. Например, сторонними производителями созданы компиляторы для языков Fortran .NET, Cobol .NET, Perl .Net — и это далеко не полный список. Все они поддерживают межъязыковую совместимость, в том числе при наследовании. Таким образом, программы для .NET Framework можно писать на любом из них — созданный код будет способен к взаимодействию с кодом для этой платформы, написанным на других языках.

Структура .NET-приложений

Чтобы понять, как общеязыковая исполняющая среда контролирует исполнение кода, необходимо изучить структуру .NET-приложения. Основным блоком .NET-приложений является *сборка* (assembly) — самоописываемый набор кода, ресурсов и метаданных. Содержимое сборки описано в ее *декларации* (manifest), которая включает следующие сведения:

- идентификационные данные сборки, такие, как имя и номер версии;
- полный список открытых типов, поддерживаемых сборкой;
- список сборок, необходимых для работы данной сборки;
- список разрешений на доступ к коду, включающий права доступа, требуемые сборкой, и права, в которых ей будет отказано.

У каждой сборки имеется только одна декларация, которая содержит все сведения, описывающие эту сборку. Декларация сборки хранится в отдельном файле либо в одном из модулей сборки.

Сборка также содержит один или несколько модулей. *Модуль* (module) хранит код приложения или библиотеки, а также метаданные, описывающие этот код. Во время компиляции проекта в сборку исходный текст, написанный на языке высокого уровня, транслируется в IL-код. Благодаря такому преобразованию свободно взаимодействуют программы на разных языках. Например, можно написать на

Visual C# приложение, которое вызывает DLL, разработанную с применением языка Visual Basic .NET. Перед исполнением оба компонента преобразуются в IL-код, что устраняет проблемы с совместимостью языков высокого уровня.

Кроме того, модули содержат типы. Тип — это шаблон кода, инкапсулирующего некоторые данные и обладающего определенной функциональностью. Видов типов два: ссылочные типы, или классы, и типы значений, или структуры. Более подробно о них рассказано на занятии 3. Для каждого типа в декларации сборки содержится описание, понятное CLR. У типа есть поля, свойства и методы, так или иначе участвующие в реализации его функциональности. Например, у класса, представляющего банковский счет, должны быть поля, свойства и методы, представляющие все функции, необходимые для реализации банковского счета. Поля предоставляют память для хранения определенного типа данных. Классу банковского счета, скорее всего, понадобится поле для хранения имени вкладчика. Свойства напоминают поля, но в отличие от них обычно проверяют значения при их установке и извлечении. В нашем примере у класса может быть свойство, хранящее остаток средств на счете. При попытке изменить значение это свойство проверит, не превышают ли изменения заданный предел, и, если это так, запретит выполнение операции. Методы представляют поведение классов, например действия, которые они выполняют над хранимыми данными, либо изменения пользовательского интерфейса. Продолжая пример, можно предположить, что у класса банковского счета будет метод *Transfer*, который переносит остаток со счета до востребования на депозит, а также метод *Alert*, извещающий пользователя о снижении остатка средств до критического уровня.

Компиляция и исполнение .NET-приложений

Как уже говорилось, во время компиляции .NET-приложений вместо двоичных машинных кодов генерируется IL-код, который представляет собой набор низкоуровневых команд, интерпретируемых CLR. Приложения развертываются в виде одной или нескольких сборок, которые состоят из исполняемых файлов и DLL, содержащих IL-код. По крайней мере одна из этих сборок должна содержать исполняемый файл, помеченный как входная точка приложения.

После запуска программы в память загружается первая сборка. Анализируя ее декларацию, CLR определяет требования для работы программы. В частности, CLR анализирует разрешения, запрашиваемые сборкой, и сравнивает их с системной политикой безопасности. Если системная политика безопасности не дает программе запрошенные разрешения, приложение не запускается, в противном случае CLR приступает к исполнению кода. Сначала CLR создает процесс, в котором будет работать приложение, и начинает исполнять его код. При этом первая порция исполняемого кода (в виде команд языка IL) загружается в память и компилируется *JIT-компилятором* (Just-In-Time compiler) в машинные коды, специфичные для данного компьютера. Машинный код исполняется и остается в памяти, поэтому каждый фрагмент кода компилируется не более одного раза во время исполнения приложения. Каждый раз, когда программа вызывает новую часть кода, которая не исполнялась ранее, JIT-компилятор заблаговременно компилирует ее и сохраняет полученный двоичный код в памяти. Такой подход позволяет достичь максимальной производительности приложения, поскольку компилируются только те части программы, которые исполняются.

Резюме

- Инфраструктура .NET Framework представляет собой основу для разработки программного обеспечения. Она состоит из общеязыковой исполняющей среды (CLR), поддерживающей основные сервисы, необходимые для исполнения программ, и библиотеки базовых классов .NET со множеством готовых классов, что значительно облегчает разработку программ. Общеязыковая спецификация (CLS) определяет минимальный набор правил, которых должны придерживаться все языки, использующие .NET Framework, а общая система типов (CTS) гарантирует совместимость типов в компонентах, написанных на разных языках.
- Основной единицей .NET-приложений считается сборка, непременным компонентом которой является декларация. Декларация описывает сборку, а также один или несколько модулей с исходным кодом приложения.
- Исполняемые файлы в .NET содержат команды языка IL. При загрузке сборки запрошенные ею разрешения проверяются на соответствие системной политике безопасности локального компьютера. Если исполнение приложения разрешается, первая сборка загружается в память и компилируется JIT-компилятором в двоичный код, непосредственно для данного компьютера; этот код остается в памяти до окончания исполнения программы.

Занятие 2. Библиотека базовых классов .NET

Библиотека базовых классов .NET — это набор объектно-ориентированных типов и интерфейсов, предоставляющих объектные модели и сервисы для решения большинства сложных задач программирования, с которыми может столкнуться разработчик. По большей части это расширяемые типы, на основе которых разработчики могут создавать собственные типы, обладающие необходимой им функциональностью. На этом занятии вы познакомитесь с некоторыми пространствами имен из библиотеки базовых классов, научитесь добавлять ссылки на нее и применять ее типы и методы.

Изучив материал этого занятия, вы сможете:

- ✓ описать библиотеку базовых классов .NET;
- ✓ объяснить, чем отличаются ссылочные типы и типы значений;
- ✓ создавать ссылки на пространства имен;
- ✓ создавать экземпляры классов и типов значений .NET Framework.

Продолжительность занятия - около 30 минут.

Библиотека базовых классов .NET Framework содержит базовые классы, представляющие большинство сервисов и объектов, необходимых для разработки приложений. Библиотека классов организована в виде пространств имен. *Пространство имен* (namespace) — это логическая группа, объединяющая типы со сходной функциональностью. Например, в пространстве имен *System.Windows.Forms* сгруппированы типы, необходимые для создания Windows-форм и размещаемых на них элементов управления.

Пространства имен, составляющие библиотеку базовых классов .NET, организованы в иерархическую структуру, корневым пространством имен которой является *System*. Пространства имен в ссылках отделяются друг от друга точками, например, так:

```
System
System.Data
System.Data.SQLClient
```

Первая конструкция из этого примера ссылается на пространство имен *System*, вторая — на пространство имен *System.Data*, а третья — на *System.Data.SQLClient*. Введение в наиболее востребованные пространства имен, содержащие базовые классы .NET, вы найдете втаблице 1-I.

Таблица **1-1. Примеры пространств имен .NET**

Имя	Описание
System	Корневое пространство имен, содержит множество низкоуровневых типов .NET Framework, в том числе основные типы данных. Поскольку это пространство имен корневое, в нем расположены все остальные пространства имен библиотеки базовых классов .NET Framework
System. Co/lections	Классы разнообразных контейнеров, такие, как *ArrayList*, *SortedList*, *Queue* и *Stack*, а также ряд абстрактных классов, полезных для создания собственных классов наборов, например *CollectionBase*

Таблица 1-1. *(окончание)*

Имя	Описание
System. ComponentModel	Классы, применяемые для создания компонентов и их контейнеров, например провайдеры лицензий, типы атрибутов и классы, выполняющие преобразование типов
System. Data	Классы, предназначенные для доступа к базам данных и работы с ними, а также дополнительные пространства имен, применяемые для доступа к данным
System. Data, Common	Набор классов, предназначенных для совместного использования управляемыми провайдерами данных из .NET
System. Data. OieDb	Классы управляемых провайдеров данных, применяемых для доступа к базам данных OLE
System. Data.SqlClient	Классы, оптимизированные для взаимодействия с базами данных SQL Server
System. Drawing	Открывает доступ к функциям GDI+ и предоставляет вспомогательные классы для формирования и визуализации графических элементов
***System.** IO*	Типы для файлового ввода-вывода
System. Math	Типы математических функций, извлекающие корни, вычисляющие тригонометрические функции и т.п.
System. Reflection	Обеспечивает получение сведений о типах и динамическое создание типов во время выполнения
System. Security	Типы, связанные с разрешениями, криптографией и защитой по правам доступа к коду
System. Threading	Классы для создания многопоточных приложений
System. Windows. Forms	Типы для создания стандартных Windows-приложений, а также классы, представляющие формы и применяемые с ними элементы управления

Можно заметить, название пространства имен отражает предоставляемую им функциональность, что не случайно. Такой подход к именованию облегчает работу с .NET Framework и позволяет быстрее освоится с ее компонентами.

Типы значений и ссылочные типы

Как говорилось ранее, в .NET Framework две разновидности типов: *ссылочные типы* (reference types) и *типы значений* (value types). Основное различие между ними заключается в способе доступа к переменным, объявленным с этими типами. Чтобы разобраться в сути этого различия, необходимо кратко пояснить основы динамического использования памяти.

Занимаемая приложением память складывается из двух основных компонентов: *стека* (stack) и *кучи* (heap). Стек — это область памяти, резервируемая для исполнения программы. Стек напоминает стопку тарелок: если нужна чистая тарелка, берут тарелку из стопки (то есть ту, которую положили последней). С переменными программы происходит то же самое: при вызове функции все объявленные в ней переменные *заталкиваются* в стек (эта операция называется *push*); если эта функция вызовет другую функцию, то все переменные, объявленные новой функцией,

тоже попадут в стек. Когда функция, вызванная последней, завершится, ее переменные выходят из *области видимости* (scope) программы и *выталкиваются* из стека (эта операция называется *pop);* в результате память, занимаемая этими переменными, освобождается, и выполнение программы продолжается.

Куча — отдельная область памяти, резервируемая для создания объектов, которые предназначены для многократного использования. CLR выделяет в куче память под объекты и управляет утилизацией памяти, занятой неиспользуемыми объектами, посредством механизма под названием *сбор мусора.*

Примечание Процедура сбора мусора обсуждается на занятии 6.

Память под экземпляры типов значений выделяется из стека. При выходе из области видимости переменной, объявленной с типом значения, переменная уничтожается, а занятая ей память освобождается. Для переменной ссылочного типа выделяются сразу две области памяти: в куче — под сам объект и в стеке — под переменную с указателем на этот объект. Когда функция обращается к такой переменной, та возвращает адрес объекта, на который она ссылается. Когда же переменная покидает область видимости, уничтожается указатель, но не сам объект. При наличии других ссылок на этот объект он сохраняется, а если все ссылки на объект уничтожены, то он становится добычей сборщика мусора (см. занятие 6).

В качестве примеров типов значений можно привести элементарные типы данных: целочисленные (*int*), булевы (*bool*), символьные *(char)* и т.п., а также структуры *(struct)* и перечислимые типы (*enum*), определяемые пользователем. Большинство ссылочных типов представлено классами. Кроме того, к ссылочным типам относятся интерфейсы, делегаты и массивы. Классы и структуры обсуждаются на занятии 3 этой главы, а остальные ссылочные типы и типы значений — в главе 3.

Примечание Везде в этой книге, где в одном месте показаны примеры ключевых слов Visual Basic и Visual C#, первым приводится пример на Visual Basic, а за ним в скобках — пример на Visual C#.

Применение типов .NET Framework в приложениях

В начало кода любого .NET-приложения автоматически помещается ссылка на библиотеку базовых классов .NET Framework, чтобы приложение «узнало» о библиотеке базовых классов и смогло создавать экземпляры типов, которые эта библиотека поддерживает.

Типы значений

Для объявления переменных с типом значений в программах на Visual Basic .NET применяется оператор *Dim;* чтобы объявить переменную в программе на C#, необходимо сначала указать тип, а затем — имя переменной, например, так:

Visual Basic .NET

```
Dim myInteger as Integer
```

Visual C#

```
int myInteger;
```

Эта команда заставляет исполняющую среду выделить память, необходимую для хранения целочисленной переменной, но не присваивает ей значение. Для определения значений переменных используется оператор присваивания:

Visual Basic .NET

```
myInteger - 42
```

Visual C#

```
myInteger = 42;
```

При желании можно инициализировать переменную при ее объявлении:

Visual Basic .NET

```
Dim myInteger as Integer - 42
```

Visual C#

```
int myInteger = 42:
```

Примечание В Visual Basic 6 не рекомендуется объявлять и инициализировать переменную в одной и той же строке, однако в Visual Basic .NET этот прием не снижает производительность.

Ссылочные типы

Экземпляры ссылочных типов создаются в два этапа. Вначале объявляется переменная требуемого типа. При этом выделяется память, необходимая для хранения данной переменной, но сам объект пока не создается. Синтаксис объявления объектов таков:

Visual Basic .NET

```
Dim myForm As System.Windows.Forms.Form
```

Visual C#

```
System.Windows.Forms.Form myForm;
```

Предыдущая строка приказывает исполняющей среде выделить необходимый объем памяти под переменную, которая в данном примере представляет форму, присваивает этой переменной имя myForm, но пока не создает в памяти реальный объект формы. Собственно объект создается в ходе второго этапа, называемого *созданием экземпляра* (instantiation), например:

Visual Basic .NET

```
myForm = New System.Windows.Forms.Form ()
```

Visual C#

```
myForm - new System.Windows.Forms.Form ();
```

Ключевое слово *New* (*new*), использованное в этом примере, вызывает конструктор типа *System. Windows. Forms.* Конструктор — это специальный метод, вызыва-

емый только в начале жизни объекта и содержащий код, который необходимо исполнить, чтобы получить работоспособный объект. Этот код способен, например, назначать свойствам объекта начальные значения. Любые параметры, необходимые конструктору, указываются в скобках в конце строки. Следующий пример иллюстрирует объявление и создание экземпляра класса *Widget*, конструктор которого требует передать ему в качестве параметра строку. Подробнее о конструкторах рассказывается на занятии 4.

Visual Basic .NET

```
Dim myWidget as Widget
myWidget = New Widget ("This string is required by the constructor")
```

Visual C#

```
Widget myWidget;
myWidget = new Widget ("This string is required by the constructor");
```

Можно объявить объект и создать его экземпляр в одной и той же строке. При этом выделяется память под объект, и в зарезервированной памяти тотчас создается экземпляр объекта. В предыдущих версиях Visual Basic совмещать объявление и создание экземпляров не рекомендуется, так как подобная экономия существенно снижает производительность. Однако Visual Basic .NET и Visual C# оптимизированы для подобных операций и выполняют их без потери производительности. Вот пример кода, иллюстрирующий объявление новой формы с одновременным созданием ее экземпляра:

Visual Basic .NET

```
Dim myForm As New System.Windows.Forms.Form ()
```

Visual C#

```
System.Windows.Forms.Form myForm = new
 System.Windows.Forms.Form ();
```

Любые переменные — как ссылочных типов, так и переменные с типами значения — необходимо инициализировать перед использованием. Поля классов и структуры в Visual Basic .NET инициализируются значениями по умолчанию при объявлении. Переменные числовых типов (например типы значений, представляющие целые числа или числа с плавающей точкой) инициализируются нулями, переменные типа *Boolean* — значением *False, a* переменные ссылочных типов — пустой ссылкой (*null*-значением).

В C# переменные по умолчанию инициализируются значением *null,* однако любая попытка использования неинициализированной переменной с классами и методами закончится ошибкой.

Применение переменных ссылочных типов и типов значений

Переменная, объявленная с типом значения, содержит все данные, которые представляет этот тип, тогда как переменная ссылочного типа содержит ссылку на некоторый объект — это принципиальное отличие между типами значения и ссылочными типами. Рассмотрим следующий пример;

Visual Basic .NET

```
Dim x, y As Integer
x = 15
y = x
x = 30
' Каково значение y?
```

Visual C#

```
int x, y;
x = 15;
y = x;
x = 30;
// Каково значение y?
```

Здесь создаются две целочисленные переменные: x и *y*. Переменной x присваивается значение 15, после чего ее значение присваивается переменной *y*. В завершение *x* присваивается значение 30. А каково значение y? Ответ кажется очевидным: 15. Так как *x* и *y* — разные переменные, модификация одной из них никак не влияет на другую. При исполнении команды *y=x* значение переменной x копируется в *y*, между этими переменными больше нет никакой связи.

Со ссылочными типами все иначе. Рассмотрим пример, аналогичный предыдущему, но использующий переменные ссылочного типа *(Form)* вместо типа значения:

Visual Basic .NET

```
Dim x, y as System.Windows.Forms.Form
x = New System.Windows.Forms.Form ()
x.Text = "This is Form 1"
y = x
x.Text - "This is Form 2"
' Какое значение вернет y.Text?
```

Visual C#

```
System.Windows.Forms.Form x,y;
x = new System.Windows.Forms.Form ();
x.Text - "This is Form 1";
y = x;
x.Text = "This is Form 2";
// Какое значение вернет y.Text?
```

Какое значение вернет *y.Text?* На этот раз ответ не столь очевиден. Поскольку *System. Windows.Forms.Form*— это ссылочный тип, переменная *x* не содержит форму, в действительности она хранит лишь ссылку на экземпляр формы. При исполнении команды *x=y* исполняющая среда копирует эту ссылку из *x* в *y*, после чего переменная *y* ссылается на тот же экземпляр формы, что и переменная x. Поскольку обе переменные ссылаются на один и тот же экземпляр объекта, они возвращают одинаковые значения свойств объекта, который они представляют, то есть *y. Text* вернет «This is Form 2».

Операторы Imports и using

Теперь вы умеете получать доступ к типам из библиотеки базовых классов .NET Framework, ссылаясь на них по полным именам. Полное имя типа включает все пространства имен, к которым он принадлежит, например:

```
System.Windows.Forms.Form
```

Однако можно заставить среду разработки «запомнить» пространства имен, которые вы используете. Для этого применяются операторы *Imports* (в Visual Basic .NET) и *using* (в Visual C#). Эти операторы позволяют ссылаться на тип по сокращенному имени, не указывая все пространства имен, к которым он принадлежит. Так, вместо *System.Windows.Forms.Form* можно записать просто *Form*. В Visual Basic .NET оператор *Imports* следует помещать в начало кода, то есть перед любыми другими операторами (за исключением *Option*). В программах на Visual C# оператор *using* располагается перед ссылками на любые элементы пространств имен, такими, как классы или структуры. Следующий пример демонстрирует использование операторов *Imports* и *using*:

Visual Basic .NET

```
Imports System.Windows.Forms
```

Visual C#

```
using System.Windows.Forms;
```

Однако во избежание конфликта все равно придется задавать полные имена для ссылки на типы с одинаковыми именами, импортируемые из разных пространств имен. Таким образом, если вы импортируете из пространств имен *MyNameSpaceOne* и *MyNameSpaceTwo* классы *Widget*, следует ссылаться на эти классы как на *MyNameSpaceOne.Widget* и *MyNameSpaceTwo.Widget* соответственно,

В программах на C# подобные конфликты имен удается разрешать при помощи псевдонимов. *Псевдоним* (alias) позволяет назначить классу другое имя и создается при помощи ключевого слова *using*:

Visual C#

```
using myAlias = MyNameSpaceTwo.Widget;
```

Назначив некоторому классу псевдоним, применяйте его в коде для ссылки на экземпляры этого класса, например, так:

Visual C#

```
// Теперь MyNameSpaceTwo.Widget можно заменить псевдонимом
// myAlias, поэтому следующие строки эквивалентны:
MyNameSpaceTwo.Widget anotherWidget = new myNameSpaceTwo.Widget;
MyAlias anotherWidget = new myAlias:
```

Ссылки на внешние библиотеки

Наверняка вам в программах понадобятся библиотеки, которые не входят в состав .NET Framework: от сторонних разработчиков или ваши собственные. Чтобы получить доступ к внешней библиотеке классов, необходимо создать ссылку на нее.

▶ **Как создать ссылку на внешнюю библиотеку**

1. В окне Solution Explorer раскройте нужный проект и щелкните правой кнопкой узел References.
2. Из контекстного меню выберите команду Add Reference — откроется диалоговое окно Add Reference.
3. Перейдите на вкладку, где находится библиотека, ссылку на которую следует создать: .NET-библиотеки расположены на вкладке .NET, унаследованные COM-библиотеки — на вкладке COM, а локальные проекты Visual Studio — на вкладке Projects.
4. Найдите необходимую библиотеку и дважды щелкните ее, чтобы добавить к списку Selected components; в подтверждение своего выбора щелкните OK.

Резюме

- Библиотека базовых классов .NET Framework — это подборка кода, который реализует функциональность, полезную для разработки приложений; ее содержимое упорядочено при помощи пространств имен, которые содержат типы и другие пространства имен, предоставляющие средства для решения общих задач.
- Все типы делятся на ссылочные типы и типы значений: переменная, объявленная с типом значения, содержит все данные, связанные с этим типом; переменная ссылочного типа содержит указатель на экземпляр объекта этого типа.
- Предопределенные типы значений создаются путем объявления и остаются пустыми до присвоения им значения. Чтобы получить объект ссылочного типа, необходимо объявить этот тип, а затем создать его экземпляр (совмещение этих действий не ведет к потере производительности).
- Во время присваивания значения одной переменной с типом значения другой переменной аналогичного типа выполняется копирование данных из первой переменной во вторую. При присваивании значения переменной ссылочного типа другой такой же переменной копируется только ссылка на объект. В результате получаются переменные, ссылающиеся на один и тот же объект.
- Операторы *using* и *Imports* позволяют ссылаться на члены пространств имен по сокращенным именам, в результате чего нет необходимости указывать их полные имена. Для использования внешней библиотеки требуется создать ссылку на нее.

Занятие 3. Классы и структуры

Итак, вы узнали, что библиотека базовых классов .NET Framework предоставляет богатый набор стандартных типов, облегчающих разработку приложений. Разработчики получают возможность создавать собственные (пользовательские) типы, реализующие нестандартные функции. На этом занятии вы познакомитесь с классами и структурами — основными разновидностями пользовательских типов.

Изучив материал этого занятия, вы сможете:

- ✓ создавать новые классы и структуры;
- ✓ создавать экземпляры классов и структур;
- ✓ уяснить различия между классами и структурами;
- ✓ создавать вложенные типы.

Продолжительность занятия - около 30 минут.

Можно сказать, что классы — это шаблоны для объектов. Класс указывает, сколько и каких данных содержит объект, но сам никаких данных не содержит. В реальном мире аналогом классов можно считать абстрактные понятия и собирательные образы. Возьмем, например, понятие «автомобиль» — оно обозначает вообще любой автомобиль. Каждый знает, что у автомобиля есть двигатель, четыре колеса, он характеризуется цветом, расходом топлива и рядом других параметров. Хотя понятие «автомобиль» включает все эти свойства, а также способность выполнять ряд действий (например, двигаться, светить фарами и т.д.), оно не обозначает никакой *конкретный* автомобиль. Понятие «автомобиль» аналогично, например, классу *Car* (по-английски «автомобиль» — «car»), тогда как объектом этого класса можно считать ваш личный автомобиль с его характеристиками — цветом, расходом топлива, типом двигателя и колесами. Характеристики машины вашего друга могут отличаться (как и объекты — значениями свойств), но он все равно останется автомобилем (объектом класса *Car*).

Члены

Классы описывают свойства и поведение представляемых ими объектов при помощи *членов* (members). Членами класса являются все его методы, поля, свойства и события. Поля и свойства содержат сведения об объекте — для автомобиля это цвет, расход топлива, тип коробки передач (автоматическая или ручная) и т.п. Методы представляют действия, которые способен выполнять объект (например, автомобиль может ехать прямо, светить фарами и много чего еще), а события — нечто интересное, происходящее с этим объектом (скажем, у автомобиля перегрелся двигатель или он попал в аварию).

Примечание В этой главе мы рассмотрим только поля и методы, а разговор о свойствах и событиях отложим до главы 3.

Создание классов

Новые классы создаются при помощи ключевого слова *Class* (Visual Basic .NET) или *class* (C#), например, так:

Visual Basic .NET

```
Public Class Widget
' Сюда помещают реализацию членов класса.
End Class
```

Visual C#

```
public class Widget
{
// Сюда помещают реализацию членов класса.
}
```

В этом примере ключевое слово *Class* (*class*) применяется для создания пользовательского класса с именем *Widget,* а ключевое слово *Public* определяет уровень доступа для этого класса. Более подробно об уровнях доступа рассказано на занятии 5.

Создание структур

Создание структур весьма напоминает создание классов. Для этой цели применяют ключевое слово *Structure* (Visual Basic .NET) или *struct* (C#), например:

Visual Basic .NET

```
Public Structure Vector
' Сюда помещается реализация структуры.
End Structure
```

Visual C#

```
public struct Vector
{
// Сюда помещается реализация структуры.
}
```

Добавление членов

В Visual Basic .NET к классу относится все, что располагается между ключевыми словами *Class* и *End Class*, а в C# — все, что заключено между фигурными скобками; то же верно и для структур. К классу или структуре разрешается добавлять члены, помещая их между указанными границами. Ниже показан пример класса *Widget,* к которому добавлен член — поле:

Visual Basic .NET

```
Public Class Widget
 Public As Integer Spin
End Class
```

Visual C#

```
public class Widget
{
```

```
 public int Spin;
}
```

Теперь у класса *Widget* есть переменная-член *Spin,* она объявлена как открытая *(Public)* и способна хранить целые числа (int). Добавление к классам других членов, таких, как методы, обсуждается на занятии 4.

Вложенные типы

Внутри типов могут быть другие типы, которые называют *вложенными* (nested). Если говорить о классах, то вложенный класс обычно представляет объект, который создается и используется родительским классом, внешний класс никогда не создает вложенный объект независимо от его родительского класса. Рассмотрим, например, класс *Wheel* («колесо»). Ему необходим набор вложенных объектов *Spoke* («спица») для внутреннего пользования. Однако пользователям класса *Wheel* никогда не потребуется создавать объект *Spoke* отдельно от объекта *Wheel.* А вот более конкретный пример. Допустим, существует класс *AccountManager,* контролирующий все особенности взаимодействия с объектами *Account,* при этом никому кроме *AccountManager* не разрешено создавать объекты *Account.* Чтобы реализовать такой сценарий, сделайте класс *Account* вложенным в *AccountManager.* Однако это не означает, что внешние объекты не смогут создать экземпляр родительского объекта — это зависит от уровней доступа к родительскому и вложенному классам. Подробнее об этом рассказано на занятии 5, а пока взгляните на пример вложенного класса;

Visual Basic .NET

```
Public Class Widget
 Код класса Widget помещается здесь,
 Private Class Widgurt
  а здесь - код класса Widgurt.
 End Class
EndClass
```

Visual C#

```
public class Widget
{
// Код класса Widget помещается здесь,
 private class Widgurt
 {
  // а здесь - код класса Widgurt.
 }
}
```

Создание экземпляров пользовательских типов

Процедура объявления пользовательских типов и создания их экземпляров не отличается от таковой для типов из .NET Framework. Чтобы задействовать в своем коде тип значения (структуру) или ссылочный тип (класс), сначала объявите переменную с этим типом, а затем создайте экземпляр этого типа при помощи ключевого слова *New (new),* например, так:

Visual Basic .NET

```
Public Class Demo
 Public Structure ValueDemo
  Public X As Integer
 End Structure
 Public Class RefDemo
  Public Y As Integer
 End Class
 Public Sub InstantiateTypes ()
  ' Объявить переменную типа ValueDemo.
  Dim DemoStructure As ValueDemo
  ' Создать экземпляр типа ValueDemo в стеке.
  DemoStructure - New ValueDemo ()
  ' Переменная готова.
  DemoStructure.X = 15
  ' Следующая строка обьявляет переменную типа RefDemo,
  ' но не создает экземпляр этого класса.
  Dim DemoClass As RefDemo
  ' А теперь можно создать объект класса RefDemo. . .
  DemoClass = New RefDemo ()
  ' ...и присвоить значения его членам.
  DemoClass.Y = 15
 End Sub
End Class
```

Visual C#

```
public class Demo
{
 public struct ValueDemo
 {
  public int X;
 }
 public class RefDemo
 {
  public int Y;
 }
 public void InstantiateTypes ()
 {
  // Объявить переменную типа ValueDemo.
  ValueDemo DemoStructure;
  // Создать экземпляр типа ValueDemo а стеке.
  DemoStructure = new ValueDemo ();
  // Переменная готова.
  DemoStructure.X = 15;
  // Следующая строка объявляет переменную типа RefDemo,
  // но не создает экземпляр этого класса.
```

```
    RefDemo DemoClass;
    // А теперь можно создать объект класса RefDemo...
    DemoClass = new RefDemo ();
    // ...и присвоить значения его членам.
    DemoClass.Y = 15;
  }
}
```

Классы и структуры

На первый взгляд классы и структуры очень похожи: у тех и других могут быть члены (например поля и методы), тем и другим требуется конструктор для создания экземпляров и, подобно всем типам NET Framework, они происходят от базового типа *Object* Ключевое отличие между ними в том, что классы — это ссылочные типы, а структуры являются типами значения. С точки зрения низкоуровневых механизмов это означает, что под экземпляры классов (объекты) память выделяется из кучи, а под экземпляры структур — из стека. Благодаря особенностям реализации, доступ к стеку осуществляется быстро и просто, но в стек нельзя поместить много данных без риска снизить общую производительность приложения.

На практике это означает, что структуры лучше применять для реализации простых небольших объектов, экземпляры которых содержат сравнительно немного данных, либо для хранения короткоживущих объектов. Классы же лучше подходят для объектов, которые содержат больше данных, либо должны храниться в памяти достаточно долгое время.

Резюме

- К пользовательским типам относятся структуры и классы. У тех и других имеются члены — поля, свойства, методы и события. Классы — это ссылочные типы, а структуры — типы значения.
- Для создания классов вVisual Basic .NET применяют ключевое слово *Class,* а в C# — *class.* Для создания структур в Visual Basic .NET служит ключевое слово *Structure,* в C# для этой цели применяют ключевое слово *struct.* Как для классов, так и для структур разрешены вложенные типы.
- Процедура объявления и создания экземпляров пользовательского типа не отличается от таковой для предопределенных типов за исключением необходимости использования ключевого слова *New (new)* для создания экземпляров пользовательских типов значений и ссылочных типов.

Занятие 4. Методы

Методы делают всю работу, для исполнения которой предназначены классы и структуры: вычисляют значения, изменяют данные, получают вводимую информацию — в общем, выполняют все действия, составляющие поведение объекта. На этом занятии вы научитесь объявлять методы, применять параметры, а также создавать конструкторы и деструкторы для своих классов.

Изучив материал этого занятия, вы сможете:

- ✓ создавать новые методы;
- ✓ задавать типы значений, возвращаемых методами;
- ✓ указывать входные и выходные параметры метода;
- ✓ создавать конструкторы и деструкторы для классов.

Продолжительность занятия - около 45 минут.

Объявление методов

Один из видов членов, которые добавляются к классам, — это методы, определяющие действия, которые должен выполнять класс. Есть две разновидности методов: те, что возвращают значения (в Visual Basic .NET они называются функциями), и те, которые никаких значений не возвращают (такие методы в Visual Basic .NET называются процедурами). Следующий пример иллюстрирует обе разновидности методов:

Visual Basic .NET

```
Public Sub MySub ()
 MessageBox.Show ("This is a non-value returning method")
End Sub

Public Function Add (ByVal first as Integer, ByVal second as _
 Integer)
 Dim Result as Integer
 Result = first + second
 Returu Result
End Function
```

В Visual C# методы подобным образом не делятся, этот язык требует указывать тип возвращаемого значения для всех методов. Если метод ничего не возвращает, при объявлении для него указывают *void* как тип возвращаемого значения. Вот пример объявления методов на C#:

Visual C#

```
public void myVoidMethod C)
(
 MessageBox.Show ("This method doesn't return a value");
)
public int Add (int first, int second)
{
```

```
 int Result;
 Result = first + second;
}
```

Вызов методов

Метод не выполняется, пока его не вызовут. Чтобы вызвать метод, следует указать его имя и задать необходимые ему параметры, например, так:

Visual Basic .NET

```
' Так вызывают метод Rotate с двумя параметрами
Rotate (45, "Degrees")
```

Visual C#

```
// Так вызывают метод Rotate с двумя параметрами
Rotate (45, "Degrees");
```

Метод *Main* — особый, он вызывается сразу после начала исполнения программы. К особым методам также относятся деструкторы, которые вызывают во время выполнения непосредственно перед уничтожением объекта. Третий вид особых методов — конструкторы; они исполняются во время инициализации объектов, о них мы поговорим чуть позже.

Переменные методов

Говорят, что область видимости переменных, объявленных внутри метода, ограничена этим методом. Это означает, что по завершении исполнения метода они уничтожаются, а занятая ими память освобождается. О таких переменных говорят, что они *вышли из области видимости.*

Область видимости переменных, объявленных внутри структур метода, еще более ограничена. Например, область видимости переменных, объявленных внутри цикла *For–Next*(в C# — *for),* ограничена этим циклом, то есть они доступны лишь в его пределах. Вот пример:

Visual Basic .NET

```
Public Sub myMethod ()
 Dim X as Integer
 For X = 1 to 100
  Dim Y as Integer
  Y = X
 Next X
  ' Эта строка вызовет ошибку
 Console.WriteLine (Y.ToString)
End Sub
```

Visual C#

```
public void myMethod ()
{
 int X;
```

```
for (X = 1; X < 101; X++)
{
 int Y;
 Y = X;
}
// Эта строка вызовет ошибку
Console.WriteLine (Y.ToString ());
}
```

Visual Basic позволяет объявлять внутри метода переменные, которые не уничтожаются по завершении исполнения этого метода. Такие переменные называются статическими переменными метода, они остаются в памяти, сохраняя свое значение, в течение нескольких вызовов метода. Чтобы сделать переменную статической, необходимо при ее объявлении указать ключевое слово *Static,* например, так:

Visual Basic .NET

```
Public Sub myMethod ()
 Static Iterations as Integer
  ' При каждом вызове метода значение этой
  ' переменной будет увеличиваться на 1.
 Iterations += 1
End Sub
```

Статическая переменная метода присутствует в памяти постоянно, но доступна только во время выполнения метода, в котором она объявлена. Такие переменные позволяют отслеживать, сколько раз вызывался метод.

Примечание В C# таких переменных нет. В этом языке имеется ключевое слово *static,* но у него совсем другой смысл, подробнее о нем рассказано на занятии 5.

Параметры

Метод может иметь один или несколько параметров. Параметр — это аргумент, передаваемый методу другим методом, который его вызывает. При объявлении метода список параметров указывают после его имени в круглых скобках, при этом для каждого параметра необходимо указать его тип. Вот пример метода с параметрами:

Visual Basic .NET

```
Public Sub DisplayName (ByVal name As String, ByVal age As Byte)
 Console.WriteLine ("Hello " & name & ". You are " & _
  age.ToString & " years old.")
End Sub
```

Visual C#

```
public void DisplayName (string name, byte age)
{
 Console.WriteLine ("Hello " + name + ". You are " + age.ToString () +
"years old.");
}
```

У этого метода два параметра: один типа *String,* с локальным именем *name,* а второй, *age,* — типа *Byte.* Их область видимости ограничена методом, вне которого переменные не доступны. Подробнее об области видимости — на занятии 5.

Есть два способа передачи параметров — *по значению* и *по ссылке.* По умолчанию в .NET Framework параметры передаются по значению. Это означает, что когда переменная передается как параметр, создается новая переменная того же типа, в которую копируется значение исходной переменной. Вызываемому методу передается новая переменная, поэтому никакие изменения, вносимые методом в переданное ему значение, не затрагивают исходную переменную. Хотя этот способ передачи параметров принят по умолчанию, в Visual Basic .NET разрешается явно указать его при помощи ключевого слова *ByVal.*

Если параметр передают по ссылке, то вместо значения переменной методу передается ссылка на область памяти, в которой оно хранится. Если вызванный метод изменит значение, переданное ему как параметр, фактически он изменит значение исходной переменной. Чтобы указать, что параметр передается по ссылке, в Visual Basic .NET применяют ключевое слово *ByRef,* а в Visual C# — *ref.* Далее показаны примеры передачи параметров по ссылке и по значению:

Visual Basic .NET

```
Public Sub Demo1 ()
 Dim x, y As Integer
 x = 15
 y = 20
 ' Вызвать метод Demo2, объявленный ниже.
 Demo2 (x, y)
 ' Каковы значения x и y?
 MessageBox.Show ("X = " & x.ToString & "Y = " & y.ToString)
End Sub
Public Sub Demo2 (ByVal p1 As Integer, ByRef p2 As Integer)
 p1 = p1 + p2
 p2 = p2 + p1
End Sub
```

Visual C#

```
public void Demol (J
{
 int x,y;
 x = 15;
 У = 20;
 // Вызвать метод Demo2, объявленный ниже.
 Demo2 (x,ref y);
 // Каковы значения x и y?
 System.Windows.Forms.MessageBox.Show ("X= " + x.ToString () +
  "Y = " + y.ToString ());
}
public void Oemo2 (int p1, ref int p2)
{
```

```
p1 = p1 + p2;
p2 = p2 + p1;
}
```

Здесь создаются две переменные — *x* и *y,* которым присваиваются начальные значения. После этого они передаются другому методу в качестве параметров, причем *x* передается по значению, а *y* — по ссылке. В вызванном методе *x* и *y* представлены соответственно переменными *p1* и *p2.* Поскольку параметр *p1* передан по значению, это просто копия переменной *x,* и все действия, выполняемые над *p1,* никак не влияют на значение переменной *x.* После завершения метода *Demo2* переменная *p1* выходит из области видимости, а занятая ею память освобождается. Иначе обстоит дело с параметром *p2: в* нем нет, собственно, значения — только ссылка на область памяти, в которой хранится значение переменной *y,* поэтому при выполнении команды *p2 = p2 + p1* изменяется значение, на которое указывает *p2.* Таким образом, после завершения метода *Demo1* значение *x* не меняется (15), а значение *y* станет равно 55.

Заметьте, что для параметра ссылочного типа не имеет значения, как он передается — по ссылке или по значению, результат окажется одним и тем же. В обоих случаях все действия, выполненные над параметром, отразятся на объекте, который был передан в качестве параметра.

Выходные параметры

В Visual C# допускается применение выходных параметров, отсутствующих в Visual Basic .NET. Выходной параметр — это тот, который вызванный метод передает (возвращает) вызвавшему его методу. Выходные параметры удобно использовать в методах, которые должны возвращать несколько значений.

Выходные параметры обозначаются ключевым словом *out,* всегда передаются по ссылке и не требуют инициализации перед использованием. Вот пример использования выходных параметров:

Visual C#

```
public void aWord (out string Word)
{
 Word = "Mambo";
}
public void ShowWord ()
{
 string Word;
 aWord (out Word);
 Console.Writeline ("The word of the day is " + Word);
}
```

Здесь метод *ShowWord* вызывает метод *aWord,* у которого есть один выходной параметр Word. Присваивая значение выходному параметру Word, метод *aWord* тем самым устанавливает значение переменной Word метода *ShowWord.*

Необязательные параметры

В Visual Basic .NET у метода могут быть необязательные параметры, а в Visual C# это не допускается. Необязательные параметры обозначаются ключевым словом

Optional и замыкают список параметров в объявлении метода. Кроме того, для необязательных параметров необходимо определить значения по умолчанию. Вот пример, иллюстрирующий использование необязательных параметров:

Visual Basic .NET

```
Public Sub Cook (ByVal time as Integer, Optional ByVal temp As _
 Integer = 350)
' Сюда помещают реализацию метода
End Sub
```

Конструкторы и деструкторы

Конструктор — это первый метод, который исполняется при создании экземпляра типа. В Visual Basic .NET конструктор всегда называется *Sub New,* а в Visual C# имя метода-конструктора совпадает с именем класса. Конструкторы служат для инициализации экземпляра класса или структуры перед его использованием и не возвращают никаких значений. Переопределив конструктор, вы заставите его выполнять нестандартную инициализацию (переопределение методов подробно описано в главе 4). Конструктор также способен вызывать другие методы. Вот пример конструктора:

Visual Basic .NET

```
Public Class aClass
 Public Sub New ()
  ' Сюда помещается инициализирующий код
 End Sub
End Class
```

Visual C#

```
public class aClass
{
 public aClass ()
 {
  // Сюда помещается инициализирующий код
 }
}
```

Деструктор (в Visual Basic он называется finalizer) выполняется последним из методов класса и содержит код, выполняющий «очистку» при уничтожении объекта класса (например, уменьшающий значения счетчика или освобождающий ресурсы). В Visual Basic .NET деструктору всегда назначается имя *Sab Finalize ()*, а в Visual C# имя метода-деструктора образуется из имени класса, предваренного тильдой (~). Вот примеры деструкторов:

Visual Basic .NET

```
Public Class aClass
 Protected Overrides Sub Finalize ()
  ' Сюда помещают код для очистки
```

```
End Sub
End Class
```

Visual C#

```
public Class aClass
!
 ~aClass ()
 {
  // Сюда помещают код для очистки
 }
\
```

Примечание В Visual Basic .NET деструкторы необходимо объявлять с ключевым словом *Overrides,* назначение и использование которого описаны в главе 4.

Недетерминированный характер сбора мусора (см. ниже) не позволяет предсказать момент, когда будет вызван деструктор класса.

Резюме

- Методы, выполняющие различные действия над данными, формируют поведение класса или структуры, где они объявлены. Методы способны (но не обязаны) возвращать значения. В Visual Basic .NET методы, возвращающие значения, называются *функциями, а* методы, которые ничего не возвращают, — *процедурами.* В Visual C# считается, что метод, не возвращающий значений, возвращает значение типа *void.* Чтобы вызвать метод, укажите в коде имя метода с необходимыми ему параметрами (в скобках).
- У методов могут быть параметры — значения, необходимые для работы метода. По умолчанию параметры передают по значению. Чтобы передать параметр по ссылке, необходимо объявить его с ключевым словом ге/(в Visual C#) или *ByRef* (в Visual Basic .NET). Параметры ссылочного типа всегда, со всем вытекающими последствиями передаются по ссылке, независимо от объявленного способа передачи. В Visual C# у метода могут быть выходные параметры, а в Visual Basic .NET — необязательные параметры.
- Конструктор — это первый метод класса, вызываемый при создании его экземпляра. Конструктор позволяет задавать начальные значения данных и выполнять другие действия до того, как объект станет доступным для использования. Деструкторы вызываются непосредственно перед уничтожением объекта, их код выполняет необходимую «очистку». Управлять вызовом деструктора невозможно.

Занятие 5. Область видимости и уровни доступа

От уровня доступа зависит, как будут создаваться экземпляры типа и осуществляться доступ к его членам. Уровни доступа применяются для инкапсуляции данных и методов типа, а также для предоставления внешним объектам доступа к функциональности типа. На этом занятии вы узнаете, как модификаторы доступа управляют доступом к коду, и научитесь применять их при разработке собственных типов.

Изучив **материал** этого занятия, **вы** сможете:

- ✓ объяснить назначение различных уровней доступа и рассказать, как они влияют на доступ к классам, переменным и вложенным типам;
- ✓ объяснить, что такое область видимости и как она влияет на исполнение программы.

Продолжительность занятия - около 20 минут.

Модификаторы доступа позволяют управлять доступом к элементам приложения. Модификаторы доступа — это ключевые слова, такие, как *Public (public), Private (private)* или *Friend (internal),* которые размешаются перед объявлением переменной или типа и управляют уровнем доступа к членам. Модификатор доступа, размещенный перед объявлением некоторого члена, задает его *область видимости (scope),* то есть определяет, какой код получит доступ к члену. Модификатор доступа, помещенный перед объявлением типа, управляет созданием экземпляров этого типа и областью видимости его членов.

Модификаторы доступа к членам

У членов типа также могут быть модификаторы, управляющие их областью видимости. Эти модификаторы перечислены в таблице 1-2.

Таблица 1-2. Модификаторы и задаваемые ими уровни доступа

Модификатор	Уровень доступа к членам
Public (Visual Basic .NET) *public* (Visual C#)	Члены доступны из любого места приложения
Private (Visual Basic .NET) *private* (Visual C#)	Доступны только членам этого же типа (в котором определен данный член)
Friend (Visual Basic .NET) *internal* (Visual C#)	Доступны из любого типа данной сборки, **но не** из внешних сборок
Protected (Visual Basic .NET) *protected* (Visual C#)	Доступны только членам этого же типа и типов, производных от него
Protected Friend (Visual Basic .NET) *protected internal* (Visual C#)	Доступны любым типам данной сборки, а также типам, производным от типа, которому принадлежит данный член; является гибридом *Protected (protected)* и *Friend (internal)*

Любой член, объявленный с модификатором *Public (public),* видим для любого кода, расположенного за пределами класса, и является открытым. Открытые поля и методы могут быть вызваны другими объектами. Напротив, методы с модифика-

тором *Private (private)* видимы только внутри типа, в котором они объявлены, их нельзя вызвать извне. Третий модификатор, *friend (internal)*, указывает, что данный член доступен типам, объявленным внутри этой сборки. Однако для типов, расположенных вне сборки, доступ к этому типу запрещен. Модификатор *Protected (protected)* разрешает доступ к члену из типа, к которому данный член принадлежит, а также из любых типов, производных от него. По правам доступа модификатор *Protected Friend (protected internal)* представляет собой гибрид соответствующих модификаторов.

При объявлении членов-переменных модификатор доступа может заменять оператор *Dim*. Если используется оператор *Dim (в* Visual Basic .NET) либо модификатор доступа не указан (в Visual C#), считается, что для переменных классов Visual Basic .NET и Visual C# задан модификатор *private,* для переменных структур Visual Basic .NET — модификатор *Public*, а для переменных структур Visual C# — *private*. Можно не указывать модификатор доступа при объявлении метода, при этом считается, что для методов классов и структур Visual C# задан модификатор *private*, а для методов классов и структур Visual Basic .NET — *Public*.

Примечание Детально наследование обсуждается в главе 4.

Следующий пример иллюстрирует применение модификаторов доступа и их действие:

Visual Basic .NET

```
Public Class aClass
 ' Это поле доступно любому внешнему коду.
 Public anInteger As Integer

 ' Этот метод могут вызвать только члены данного
 ' класса или данной сборки, внешнему коду он не доступен.

 Friend Sub myMethod ()
 End Sub

 ' Это поле доступно только членам данного класса.
 Private aString As String

 ' Этот метод могут вызывать только члены данного класса и
 ' классов, производных от него.
 Protected Function Return1 () As Integer
  Return 1
 End Function

 ' Это поле доступно членам любых типов из данной сборки,
 ' а также типов, производных от данного класса.
 Protected Friend aLong As Long

End Class
```

Visual C#

```
public class aClass
{
// Это поле доступно любому внешнему коду.
public int anInteger;

// Этот метод могут вызвать только члены данного
// класса или данной сборки, внешнему коду он не доступен.

internal void myMethod ()
{
}

// Это поле доступно лишь членам данного класса
private string aString;

// Этот метод могут вызывать только члены данного класса и
// классов, производных от него
protected int Return1 ()
{
 return 1;
}

// Это поле доступно членам любых типов из данной сборки,
// а также типов, производных от данного класса.
protected internal long aLong;

}
```

Модификаторы доступа к типам

Доступом к структурам и классам также можно управлять средствами модификаторов, контролирующих создание экземпляров; они напоминают модификаторы, управляющие доступом к членам. Экземпляр класса, объявленного с модификатором *Public (public)* способен создать любой объект приложения, объект класса с модификатором *Friend (internal)* — любой объект сборки, а объекты, которые не входят в данную сборку, этого сделать не могут. Модификаторы *Private (private)* и *Protected (protected)* применяются только с вложенными типами. Объект класса, объявленного с модификатором *Private (private),* создается только объектами того же типа или объектами его родительского типа (то есть типа, который содержит этот вложенный тип). Вложенные типы с модификатором *Protected (protected)* или *Protected Friend (protected internal)* доступны классам, производным от их родительского класса. Классы, объявленные с модификатором *Protected friend (protected internal),* также видимы всем классам из данного пространства имен. Если модификатор доступа для класса или структуры не указан, по умолчанию считается, что задан модификатор *Public (public).*

Примечание Более подробно защищенные (protected) типы рассматриваются в главе 4.

Модификаторы доступа для вложенных типов

Вложенный тип, как правило, используется только его родительским типом. Поэтому для вложенных типов лучше всего подходит модификатор доступа *Private (private)*. Другие модификаторы нужны только для вложенных типов, допускающих создание своих экземпляров типами, отличными от родительского, что требуется довольно редко. В принципе, у вложенного типа может быть любой модификатор доступа, но его область видимости никогда не будет шире таковой его родительского типа. Рассмотрим следующий пример;

Visual Basic .NET

```
Friend Class ParentClass
 Public Class NestedClass
 End Class
End Class
```

Visual C#

```
internal class ParentClass
{
 public class NestedClass
 {
 }
}
```

Хотя вложенный класс *NestedClass* объявлен здесь как открытый [*Public (public)*], он находится внутри класса с модификатором *Friend (internal),* поэтому его не увидит ни один класс, расположенный за пределами данной сборки. Дело в том, что его родительский класс помечен модификатором *Friend (internal),* поэтому вложенному классу автоматически назначается такой же уровень доступа.

Общие (статические) члены

Обычно члены экземпляра класса принадлежат только данному экземпляру и никому другому, что демонстрирует следующий псевдокод:

Visual Basic .NET

```
Dim Object1 as New DemoClass ()
Dim Object2 as New DemoClass ()
Object1.MyField = 15
Object2.MyField = 20
```

Visual C#

```
DemoClass Object1 = new DemoClass ();
DemoClass Object2 = new DemoClass ();
Object1.MyField = 15;
Object2.MyField = 20;
```

Поле *MyField* разных объектов класса *DemoClass* содержит разные значения. Однако можно создавать члены, общие для всех экземпляров класса. Они называются

общими *(Shared)* или статическими (*static*). Независимо от числа экземпляров типа возможно не более одного экземпляра каждого из его общих членов.

Чтобы создать общее поле, необходимо объявить его с ключевым словом *Shared* (в Visual Basic .NET) или *static* (в C#), например:

Visual Basic .NET

```
Public Class Demo
 Public Shared MyField As Integer
End Class
```

Visual C#

```
public class Demo
{
 public static int MyField;
}
```

Даже при наличии нескольких экземпляров класса Demo существует только один экземпляр поля MyField. Заметьте, что ключевое слово *Shared (static)* не является модификатором доступа, оно лишь свидетельствует о том, что помеченный им член является общим. Для общих членов разрешается задавать любые модификаторы доступа — *Public (public), Private (private), Friend (internal)* и т.д.

Методы, как и поля, тоже могут быть общими (статическими). Если обычные методы принадлежат экземплярам типа, то общие методы — самому типу, поэтому общие методы не имеют доступа к данным экземпляров этого типа, откуда бы их ни вызывали. Общему методу доступны лишь общие переменные, переменные, объявленные в этом методе, а также переданные ему параметры.

Доступ к общим членам

Поскольку общие члены принадлежат типу, а не его экземпляру, на них нужно ссылаться по имени класса, а не экземпляра. Хотя Visual Basic .NET позволяет обращаться к общим членам из разных экземпляров класса, при этом вы все равно обращаетесь к одному и тому же экземпляру общего члена. В этом отношении правила, принятые в Visual C#, более строги: они не позволяют обращаться к статическим членам из экземпляров класса.

Visual Basic .NET

```
' Здесь используется класс Demo из предыдущего примера.
Dim Object1 as New Demo C)

' Обращаться к общему полю из объекта некорректно,
' хоть это и не вызывает ошибку.
Object1.MyField = 15

' Это корректный синтаксис для вызова общего поля:
' он использует имя класса, а не экземпляра.
Demo.MyField = 15
```

Visual C#

```
// Здесь используется класс Demo из предыдущего примера.
Demo Object1 = new Demo ();

// Синтаксическая ошибка! В Visual C# нельзя обращаться
// к общим членам через объект.
Object1.MyField = 15;

// Это корректный синтаксис для вызова общего поля:
// он использует имя класса, а не экземпляра.
Demo.MyField = 15;
```

Поскольку общие члены принадлежат типу, а не его экземпляру, к ним можно обращаться, не создавая экземпляр данного класса, то есть вызывать общие методы и поля типа, даже если не существует не одного его экземпляра.

Резюме

- Модификаторы доступа управляют областью видимости членов типа. Модификаторов, определяющих различные уровни доступа, всего пять: *Public (public), Friend (internal), Private (private), Protected (protected)* и *Protected Friend (protected internal).*
- Если для метода не указан модификатор доступа, то в классах и структурах Visual C# для него по умолчанию принимается уровень доступа *private,* а в классах и структурах Visual Basic .NET — *public.* Соответственно для членов-переменных по умолчанию принимаются уровни доступа *private* (в классах и структурах Visual C#) и *public* (в структурах Visual Basic .NET).
- Модификаторы доступа к типам управляют созданием экземпляров типов. С типами применяются следующие модификаторы: *Public (public)* — разрешает создавать экземпляры типа в любом месте приложения; *Friend (internal)* — позволяет создавать экземпляры типа только членам из данной сборки и *Private (private)* — разрешает создавать экземпляры только самому типу и его родительским типам.
- Если для класса или структуры модификатор доступа не задан, по умолчанию для них принимается модификатор *Public (public),*
- Вложенные типы подчиняются аналогичным правилам, однако их уровень доступа не может быть больше уровня доступа родительского класса.
- Общие (статические) члены принадлежат типу, а не его экземплярам, поэтому они доступны без создания экземпляров типа. К общим членам следует обращаться по имени класса, а не его экземпляра. Общие методы не имеют доступа к данным экземпляров типа, в котором они объявлены.

Занятие 6. Сбор мусора

Схема автоматического управления памятью, применяемая в .NET Framework, получила название *сбор мусора* (garbage collection). Этот механизм позволяет отслеживать неиспользуемые объекты и освобождать занятую ими память совершенно без участия приложения. На этом занятии вы узнаете, как работает механизм сбора мусора.

Изучив материал этого **занятия,** вы сможете:

- ✓ уяснить, как освобождается память неиспользуемых объектов при сборе мусора;
- ✓ понять, как сборщик мусора управляется с циклическими ссылками,

Продолжительность занятия - около 15 минут.

В .NET Framework применяется автоматическое управление памятью, то есть .NET Framework автоматически утилизирует память неиспользуемых объектов. Этот процесс называется сбором мусора. Рассмотрим следующий пример:

Visual Basic .NET

```
Sub GarbageCollectionExample1 C)
 Dim myWidget as New Widget ()
End Sub
```

Visual C#

```
void GarbageCollectionExample1 {)
{
 Widget myWidget = new Widget ();
}
```

Когда эта процедура завершится, переменная *myWidget* выйдет из области видимости, после чего на нее не будет ссылаться ни одна из переменных приложения. Сборщик мусора непрерывно проверяет дерево ссылок в фоновом режиме и определяет объекты, на которые больше нет ссылок. Обнаружив такой объект (например, *myWidget* из нашего примера), сборщик мусора удаляет его и освобождает занятую им память.

Поскольку сбор мусора работает постоянно, не обязательно явно уничтожать объекты, как только они станут ненужными. Обычно сборщик мусора работает в низкоприоритетном потоке, который исполняется, пока процессор не занят более важными задачами, но при дефиците памяти приоритет этого потока повышается. В результате память освобождается быстрее, пока не исчезнет ее дефицит, тогда приоритет потока сборщика мусора вновь снижается.

Подобный подход к освобождению памяти, получивший название недетерминированного, призван обеспечить максимальную производительность приложений и предоставить им окружение, более устойчивое к ошибкам. Однако за эти удобства приходится расплачиваться неопределенностью. Невозможно с уверенностью сказать, когда именно будет уничтожен тот или иной объект, то есть нельзя контролировать момент вызова деструктора класса (в C#) или метода Visual Basic .NET, подготавливающего объекты к уничтожению. В силу этих причин в деструкторе не

должно быть кода, который необходимо исполнить в заданное время. Классы, удерживающие дефицитные ресурсы, обычно вызывают метод *Dispose ()*, гарантированно освобождающий эти ресурсы, как только объект становится ненужным,

Циклические ссылки

Сборщик мусора успешно справляется с циклическими ссылками, которые раньше были одной из наиболее распространенных причин утечек памяти. Рассмотрим следующий пример:

Visual Basic .NET

```
Class Widget
 Public ChildWidget As Widget
 Public Parent As Widget
End Class
Class aClass
 Public GrandParent As Widget
 Sub Demo ()
  Dim Parent As Widget
  Dim Child As Widget
  GrandParent = New Widget ()
  GrandParent.ChildWidget = New Widget ()
  Parent = GrandParent.ChildWidget
  Parent.ChildWidget = New Widget ()
  Child = Parent.ChildWidget
  Child.Parent = Parent
  GrandParent = Nothing
 End Sub
End Class
```

Visual C#

```
class Widget
{
 public Widget ChildWidget;
 public Widget Parent;
}
class aClass
{
 Widget GrandParent;
 void Demo ()
 {
  Widget Parent;
  Widget Child;
  GrandParent = new Widget ();
  GrandParent.ChildWidget = new Widget ();
  Parent - GrandParent.ChildWidget;
  Parent.ChildWidget = new Widget ();
  Child = Parent.ChildWidget;
```

```
    Child.Parent = Parent;
    Grandparent = null;
  }
}
```

Класс *Widget* состоит из полей *ChildWidget* и *Parent*, в которых хранятся ссылки на объекты *Widget*. В этом примере создается экземпляр класса *Widget*, и ссылка на него записывается в переменную *GrandParent*. Затем этот объект создает еще один экземпляр *Widget* и записывает ссылку на него в свое поле *ChildWidget*, а также в переменную *Parent*. Далее объект *Parent* создает третий объект типа *Widget* и записывает ссылку на него в свое поле *ChildWidget* и в переменную *Child*. Наконец, в поле *Parent* объекта *Child* помещается ссылка на объект *Parent* — в результате получается обратная ссылка от объекта *Child* на объект *Parent*. После сброса переменной *GrandParent* (путем записи в нее *null*), остается пара объектов, ссылающихся только друг на друга — налицо циклическая ссылка!

При разработке с использованием других платформ циклические ссылки вызывают утечки памяти, которые очень трудно обнаружить, однако сборщик мусора .NET Framework в состоянии отследить и устранить их. Так, если обнаружена пара объектов, ссылающихся только друг на друга, оба этих объекта помечаются как мусор.

Резюме

- .NET Framework обеспечивает автоматическое освобождение неиспользуемой памяти посредством сбора мусора, который осуществляет низкоприоритетный поток, работающий в фоновом режиме. При возникновении дефицита памяти приоритет сборщика мусора повышается, пока он не освободит достаточно памяти.
- Нельзя точно предсказать, когда сборщик мусора уничтожит объект, поэтому не помещайте в деструкторы код, который требуется исполнять в заданное время, Для максимально быстрого освобождения дефицитных ресурсов следует применять метод *Dispose ()*, который разрешается вызывать явно.
- Сборщик мусора непрерывно отслеживает дерево ссылок и освобождает память, занятую объектами, на которые никто не ссылается, а также объектами, связанными циклическими ссылками.

Практикум 1. Применение классов и демонстрация сбора мусора

Цель этого практикума — получение и закрепление навыков создания классов и членов. Для этого вы напишете программу, демонстрирующую автоматическое освобождение памяти при помощи сбора мусора. Для этого следует создать класс, взаимодействующий с готовым пользовательским интерфейсом. У него будет общая переменная для учета его экземпляров, существующих в памяти. Код, увеличивающий или уменьшающий значение этой переменной, вы поместите соответственно в конструктор и деструктор этого класса. Далее вы создадите несколько экземпляров класса и посмотрите, как сборщик мусора освобождает занятую ими память. Готовую программу вы найдете на прилагаемом к книге компакт-диске в каталоге \Labs\Ch01\Solution.

Продолжительность практикума - около 20 минут.

Примечание Решение задач этого и всех остальных практикумов (на Visual Baste .NET и Visual C#) из данной книги вы найдете на прилагаемом компакт-диске в каталогах \Labs\Chxx\Solutions, где xx — номер соответствующей главы.

Упражнение 1. Создание класса Demo

Сейчас вы создадите класс *Demo,* взаимодействующий с проектом *Demo-Test* (его можно взять с прилагаемого CD из каталога \Labs\Ch01\Partial).

▶ Создание класса Demo

1. Откройте решение DemoTest.sln из каталога \Labs\Ch01\Partial — оно содержит интерфейс, необходимый для проекта.
2. В меню Project выберите Add Class.
3. В диалоговом окне Add New Item введите для класса имя **Demo.vb** или **Demo.cs**, в зависимости от используемого языка программирования.
4. Объявите в классе *Demo* открытое общее поле *Instances* (см. код ниже). Оно требуется для учета экземпляров класса *Demo,* существующих в памяти:

Visual Basic .NET

```
Public Shared Instances as Long
```

Visual C#

```
public static long Instances;
```

5. Напишите конструктор для нового класса самостоятельно, если вы используете Visual Basic .NET, или воспользуйтесь конструктором по умолчанию, генерированным Visual Studio (если вы пишете на Visual C#). В конструкторе должен быть код, увеличивающий значение переменной *Instances:*

Visual Basic .NET

```
Public Sub New ()
 Instances += 1
End Sub
```

Visual C#

```
public Demo ()
{
 Instances++;
}
```

6. Создайте для класса *Demo* деструктор с кодом, уменьшающим значение переменной *Instances*:

Visual Basic .NET

```
Protected Overrides Sub Finalize ()
 Instances -= 1
End Sub
```

Visual C#

```
~Demo ()
{
 Instances -;
}
```

Примечание В деструкторе, написанном на Visual Basic .NET, должно быть ключевое слово *Overrides* (см. главу 4).

7. В меню File выберите команду Save All, чтобы сохранить результаты своей работы.

Упражнение 2. Демонстрация сбора мусора

Интерфейс, взятый из проекта DemoTest, состоит из формы с двумя видимыми элементами управления — кнопкой (button) и надписью (label) и одним невидимым — таймером, обновляющим надпись каждые 10 секунд. Запустите созданное вами приложение и пронаблюдайте за созданием экземпляров класса Demo и их уничтожением при сборе мусора,

▶ Демонстрация работы сборщика мусора

1. Откройте форму Form1 и убедитесь, что на ней есть элементы управления Button и Label, а в области компонентов присутствует компонент Timer.

Примечание Подробнее об элементах управления — в главе 2.

2. Щелкните правой кнопкой Button1 — откроется окно кода с обработчиком события *Click* для кнопки.

3. Найдите строку *Private Sub Button1_Click* (Visual Basic .NET) или *void button1_Click* (Visual C#). Изучите следующий код:

Visual Basic .NET

```
Dim Counter As Integer
Dim aDemo
For Counter = 1 to 1000
 aDemo = New Demo ()
Next
```

Visual C#

```
int  Counter;
Demo aDemo;
for (Counter = 0; Counter < 1000; Counter++)
{
 aDemo = new Demo ();
}
```

Здесь объявляются две переменные типа *Demo* — *Counter* и *aDemo*, за которыми следует цикл, тысячу раз присваивающий переменной *aDemo* ссылку на новый экземпляр класса *Demo*. При создании каждого экземпляра класса *Demo* выполняется его конструктор, который увеличивает значение переменной *Instances*. Запись ссылки на новый экземпляр *Demo* в переменную *aDemo*, выполняемая в цикле, уничтожает ссылку на предыдущий экземпляр. С каждым оборотом цикл создает все больше объектов *Demo,* на которые никто не ссылается, увеличивая работу для сборщика мусора. Каждый щелчок кнопки создает тысячу новых объектов.

4. Нажмите F4, чтобы скомпоновать и запустить приложение. Появится кнопка и надпись, отображающая число экземпляров класса *Demo,* существующих в памяти. Щелкните кнопку один раз.

   На форме появится следующая надпись; «There are 1000 instances of Demo in memory». Подождите немного, через некоторое время (примерно через пару минут) вы увидите, что в памяти не осталось ни одного экземпляра *Demo*. Это означает, что все 1000 экземпляров класса *Demo* были удалены при сборе мусора. При удалении каждого объекта был исполнен деструктор, при этом значение переменной *Instances* уменьшалось.

   Поскольку приоритет поток сборщика мусора относительно невысок, надпись изменяется не мгновенно. Однако при нехватке памяти приоритет этого потока возрастает.

5. Щелкните кнопку несколько раз подряд. Посмотрите, сколько экземпляров вашего объекта уместится в памяти. Если на компьютере установлено много памяти, число объектов может достигать десятков тысяч, прежде чем начнется сбор мусора. Как только возникнет дефицит памяти, сборщик мусора быстро и эффективно освободит память, занятую объектами, на которые нет ссылок.

Закрепление материала

Приведенные ниже вопросы помогут вам лучше усвоить основные темы данной главы. Если вы не сумеете ответить на вопрос, повторите материал соответствующего занятия. Ответы для самопроверки — в приложении «Вопросы и ответы» в конце книги.

1. Кратко перечислите главные компоненты .NET Framework и опишите, для чего они предназначены.

2. Дайте определение ссылочным типам и типам значения.

3. Как ссылаться на члены библиотеки базовых классов .NET при помощи их сокращенных имен?

4. Расскажите, как работает сборщик мусора.

5. Что такое члены класса? Назовите четыре типа членов.

6. Что такое конструкторы и деструкторы и для чего они нужны?

7. Кратко поясните различия между уровнями доступа, которые задают для пользовательских типов и членов модификаторы *Public (public)*, *Friend (internal)* и *Private (private)*.

8. Нужно ли создавать экземпляр класса для обращения к общему члену? Почему?

ГЛАВА 2

Создание пользовательского интерфейса

Занятие 1. Принципы разработки пользовательского интерфейса 40

Занятие 2. Работа с формами 45

Занятие 3. Применение элементов управления и компонентов 58

Занятие 4. Меню 71

Занятие 5. Проверка данных, вводимых пользователем 79

В этой главе

Эта глава посвящена созданию пользовательского интерфейса для форм. Здесь рассматриваются ключевые вопросы разработки интерфейса — принципы дизайна, применение элементов управления, конструирование меню и форм, способы проверки данных, вводимых пользователем.

Прежде всего

Для выполнения занятий этой главы достаточно освоить ее материал.

Занятие 1. Принципы разработки пользовательского интерфейса

Интерфейс — это механизм, обеспечивающий взаимодействие пользователей с приложением. Рациональная конструкция пользовательского интерфейса, обеспечивающая простоту работы с ним, имеет первостепенное значение при создании приложений. На этом занятии вы познакомитесь с правилами разработки дружественных пользователю, элегантных и простых в применении интерфейсов.

Интерфейс прежде всего должен учитывать потребности конечных пользователей приложения, которых принято называть *целевой аудиторией.* Если целевая аудитория программы известна, разработка пользовательского интерфейса не представляет трудностей. Рационально сконструированный интерфейс облегчает освоение и применение программы, а неудачный вызывает у клиентов ощущение бессилия и снижает эффективность работы с приложением. В результате пользователи все чаще игнорируют его и даже могут полностью отказаться от него.

Центральный элемент интерфейса приложения для Windows — форма, Это основа взаимодействия пользователя с приложением на любом уровне. Для поддержки необходимой функциональности к приложению добавляют соответствующие элементы управления и меню. Однако кроме этого интерфейс должен быть привлекательным и вызывать у пользователя желание работать. .NET Framework поддерживает богатый набор графических эффектов, позволяющих сделать внешний вид приложения более привлекательным, в том числе фигурные формы и элементы управления, прозрачные или полупрозрачные компоненты, а также сложные эффекты затенения.

Изучив материал этого занятия, вы сможете:

- ✓ объяснить, почему пользовательский интерфейс так важен;
- ✓ рассказать о роли форм, элементов управления и меню в пользовательском интерфейсе;
- ✓ пояснить значение основных принципов дизайна — выбора оптимальной композиции и цветов;
- ✓ рассказать о применении изображений, значков и шрифтов при интерактивной разработке интерфейса.

Продолжительность занятия - около 30 минут.

Формы, элементы управления и меню

Форма — это основной элемент пользовательского интерфейса, основа для построения приложения. Обычно формы отображают логически связанные данные и содержат инструментальные средства, позволяющие пользователю получать информацию, необходимую для продолжения работы. Любая форма одновременно и класс, поэтому удается создавать экземпляры форм и порождать одни формы от других — об этом будет рассказано на занятии 2 и в главе 4.

При помощи элементов управления конечный пользователь получает необходимую информацию и работает с программой. Одни элементы управления, такие, как надписи и изображения, только отображают информацию, тогда как поля и списки, кроме того, способны к приему вводимых пользователем данных, а кнопки пре-

доставляют пользователю возможность управлять развитием событий. Применение элементов управления обсуждается на занятии 3.

Меню и панели инструментов упорядочивают команды, доступные пользователю. Как правило, меню открывает доступ к командам верхнего уровня, общим для всех форм приложения (например, командам для сохранения данных или выхода из приложения). Логично и последовательно организованное меню приносит удовлетворение от работы с приложением и помогает быстро освоить его возможности. Активация и деактивация элементов меню позволяет менять набор доступных пользователю возможностей в зависимости от того, в каком месте приложения тот находится. О применении меню рассказывается на занятии 4.

Принципы дизайна интерфейса

Дизайн пользовательского интерфейса во многом определяет впечатление от работы с приложением. От него зависит, примут ли пользователи приложение и как быстро они освоят его. Основных принципов дизайна немного:

- простота;
- оптимальное размещение элементов управления;
- логика и последовательность;
- эстетика внешнего вида.

Простота

Простота пользовательского интерфейса — это одна из важнейших его характеристик. Освоение и использование программы с запутанным или чересчур сложным интерфейсом весьма затруднительно и требует много времени. Пользовательский интерфейс должен помогать быстро исполнять любые операции, необходимые для взаимодействия с программой, предоставляя ровно столько возможностей, сколько требуется на данном этапе работы.

Конструировать интерфейс следует с учетом логики работы приложения. Элементы управления, отображающие логически связанные данные, следует размещать на формах группами. То же касается списков, полей и флажков для выбора и установки предопределенных параметров. Корректная передача фокуса позволяет быстро перемещаться между элементами управления, составляющими группу, нажимая клавишу Tab.

При разработке интерфейса часто допускают ошибку, пытаясь точно воспроизвести объект из реального мира. Например, при конструировании формы для замены бумажного бланка возникает естественное желание воспроизвести этот бланк «один к одному». Иногда такой подход оправдан, но чаще он лишь ограничивает возможности приложения, не принося реальной выгоды. В любом случае следует хорошо обдумать ситуацию и попытаться расширить функциональность приложения за счет дополнительных возможностей, которые предоставляет компьютер,

Применение значений по умолчанию также позволяет упростить пользовательский интерфейс. Если большинство клиентов выбирают значение «Вашингтон» для поля «Штат», имеет смысл назначить «Вашингтон» значением по умолчанию для этого поля. Однако следует позаботиться о том, чтобы при необходимости пользователь смог без труда указать другое значение.

Пользовательский интерфейс рекомендуется создавать на основе данных опроса целевой аудитории. Узнайте, как пользователи хотят работать с приложением, и постарайтесь спроектировать интерфейс так, чтобы упростить решение наиболее

типовых задач. Результаты опросов также пригодятся для совершенствования готовыхприложений.

Композиция формы

При размещении элементов управления следует учитывать, насколько они важны и как часто они требуются. Предположим, есть форма для ввода обязательной и необязательной информацию. Ясно, что элементы управления для ввода обязательной информации важнее и поэтому должны выделяться среди остальных элементов формы. Как правило, предполагают, что пользовательский интерфейс просматривают слева направо и сверху вниз, поэтому считается, что наиболее важные и востребованные элементы управления лучше размещать в верхней части формы. Элементы управления, с которыми пользователь работает после завершения основных действий на форме (такие, как кнопка Submit), логично разместить в нижней части формы.

Следующий вопрос — согласованность отображаемой информации. Элементы управления, показывающие логически связанные данные, следует группировать. Допустим, вы создаете форму для публикации сведений о клиенте, товаре или служашем. Для ее создания можно сгруппировать необходимые элементы управления средствами элемента *Tab,* чтобы пользователь легко перемешался между ними.

Наконец, позаботьтесь об эстетике размещения элементов управления на форме. Избегайте перегруженных форм, содержащих больше данных, чем можно усвоить с первого взгляда. Старайтесь оставлять достаточно свободного места между элементами управления — так вы придадите форме привлекательный вид и облегчите ее чтение.

Согласованность

Облик интерфейса должен гармонировать с содержанием приложения, в противном случае интерфейс раздражает взгляд, а приложение кажется нелогичным и дезорганизованным, что затрудняет его освоение.

Согласованность следует закладывать в проекте. Для этого необходимо еще до разработки создать стройную и логичную схему внешнего вида приложения. Определяя облик интерфейса, продумайте цвета, шрифты, размеры и тип используемых элементов управления. Не поддавайтесь искушению сделать интерфейс броским — применяйте лишь то, что требуется для решения поставленной задачи. Перегрузка элементами управления и множество броских, но неуместных эффектов лишь отвлекает пользователя и в конечном счете снижает производительность его труда.

Все формы приложения необходимо создавать по единой схеме. Позаботьтесь о том, чтобы пользователю не пришлось заново привыкать к размещению элементов интерфейса каждый раз, когда он открывает новую форму.

Эстетика внешнего вида

По возможности сделайте внешний вид приложения привлекательным и приятным, однако ни в коем случае не приносите ясность и простоту интерфейса в жертву привлекательности. Достаточно, чтобы интерфейс не был отталкивающим.

Выбор цветов

Грамотное использование цветов — один из действенных способов улучшить внешний вид интерфейса, однако берегитесь излишеств. Яркие кричащие цвета при-

влекают одних пользователей, но вызывают дискомфорт у других, поэтому в качестве фона лучше выбирать приглушенные тона.

При разработке цветовой схемы приложения также важно ориентироваться на целевую аудиторию. Если вы выполняете заказ компании, стоит воспользоваться ее фирменными цветами, а если программа предназначена для многонациональной аудитории, учитывайте смысловую нагрузку тех или иных цветов в различных культурах. Стремитесь к последовательности в оформлении интерфейса, но в разумных пределах.

Подумайте, будут ли выбранные цвета удобными для пользователей, в том числе для людей с дефектами зрения. Например, голубой цвет на черном фоне читается очень плохо. Не забывайте также о цветовой слепоте: некоторые люди не различают красный и зеленый цвета, поэтому они не увидят красный текст на зеленом фоне. Сделайте важные элементы приложения контрастными, но не полагайтесь при выделении информации исключительно на цвет.

Подбор шрифта

Основным критерием выбора шрифта является его удобство, причем схема применения шрифтов должна быть единой для всего приложения. Избегайте трудно читаемых и вычурных гарнитур, старайтесь применять простые, легко читаемые шрифты вроде Palatino или Times New Roman. Шрифты, имитирующие рукописный текст, а также декоративные допускаются для визуального выделения, например, заголовков страниц, но только там, где это уместно. Никогда не оформляйте такими шрифтами важный текст.

Изображения и значки

Изображения и значки делают интерфейс более привлекательным, но при этом важно тщательно продумать их применение. Излишние и отвлекающие изображения затрудняют работу с приложением. Хотя значки успешно применяются во многих приложениях как альтернативный способ представления информации, это не отменяет необходимость их тщательного выбора с учетом потребностей конечного пользователя. Например, значок в виде дорожного знака «stop», запрещающего движение, подходит для предупреждения пользователя о возможных нежелательных последствиях их действий. Однако дорожные знаки одинаковы не во всех странах, и вполне вероятно, что для некоторых ваших пользователей такой значок окажется бессмысленным. Изучайте целевую аудиторию и выбирайте значки и другие графические элементы, исходя из ее особенностей.

Лучше для значков применять простые изображения, которые легко передаются квадратом размеров 16×16 пикселов. Более сложные значки сильно искажаются при просмотре с низким разрешением.

Фигурные и прозрачные элементы управления

.NET Framework предоставляет инструменты для создания элементов управления разной прозрачности и формы, отличной от традиционного прямоугольника. Хотя это чрезвычайно мощное средство создания визуальных эффектов для интерфейса, пользуйтесь им в меру. Например, текстовое поле в форме пончика выглядит забавно и оригинально, но едва ли оно повысит эффективность приложения или сделает его удобнее. Применяя подобные эффекты, всегда помните об интересах конечного пользователя. Разнообразные элементы управления и формы (например круглые кнопки) делают пользовательский интерфейс привлекательным и не вы-

биваются из стиля приложения, только когда применены со вкусом. Полупрозрачные формы хороши тем, что позволяют держать в поле зрения форму, расположенную на заднем плане. Эстетика повышает удобство работы с приложением.

Резюме

* Дизайн интерфейса очень важен, поскольку выдержанный в едином стиле и логично организованный интерфейс облегчает освоение и эффективное использование программы.
* Главные компоненты пользовательского интерфейса — это формы, элементы управления и меню.
* Хороший пользовательский интерфейс тщательно продуман и разработан. В нем сочетаются:
  - простота;
  - оптимальное расположение элементов управления;
  - логика и последовательность;
  - эстетика внешнего вида.
* Выбирайте элементы оформления интерфейса на основе потребностей целевой аудитории вашего приложения. Однако непременно учитывайте смысловую нагрузку этих элементов в контексте различных культур. Основные принципы разработки интерфейса — последовательность и простота.

Занятие 2. **Работа с формами**

Как говорилось ранее, форма — это фундаментальная единица интерфейса. Формы представляют собой основу для взаимодействия с пользователем, «площадку» для построения гармоничного и привлекательного интерфейса. Формы получают от пользователя данные и отображают их. В принципе, возможно приложение и без единой формы (например, сервисы Windows и консольные приложения). Однако приложению, постоянно взаимодействующему с пользователем, необходима по крайней мере одна форма, а для нормальной работы более сложных приложений иногда требуется несколько форм.

Изучив материал этого занятия, вы сможете:

- ✓ описать роль форм в приложениях;
- ✓ добавить форму к приложению;
- ✓ назначить стартовую форму и выбирать ее первоначальное размещение;
- ✓ определить внешний вид формы;
- ✓ вызывать методы формы;
- ✓ применять события формы.

Продолжительность занятия - около 30 минут.

Добавление **формы к проекту**

Благодаря формам пользователь взаимодействует с приложением. При создании нового проекта Windows Forms к нему по умолчанию добавляется форма с именем Form1. В действительности это не экземпляр формы, а класс, представляющий ее объект кода. Эту форму можно редактировать средствами инструмента под названием *дизайнер* (designer), добавлять к ней элементы управления, меню и прочие графические компоненты. При усложнении приложения наступает момент, когда требуются дополнительные классы форм.

▶ **Как добавить к проекту новую форму**

1. Выберите из меню Project команду Add Windows Form — откроется диалоговое окно Add New Item.
2. Щелкните Windows Form, затем Open — в среде разработки появится новая форма.

Новую форму можно также добавить напрямую, отредактировав код. В этом случае объявляется переменная типа *Form* и создается экземпляр этого типа. Созданную таким образом форму разрешено использовать и отображать в течение всего срока исполнения приложения. Однако ее нельзя редактировать с помощью инструментов с графическим интерфейсом, так как она недоступна во время разработки. Этот прием часто применяют для отображения готовых форм.

▶ **Как добавить к проекту новую форму во время выполнения**

Объявите переменную с типом формы и создайте экземпляр формы. Это делается так же, как для любого другого класса, например:

Visual Basic .NET

```
' Этот пример предполагает наличие готовой формы
' с именем DialogForm
Dim myForm As DialogForrn
myForm = New DialogForm()
```

Visual C#

```
// Этот пример предполагает наличие готовой формы
// с именем DialogForm
DialogForm myForm;
myForm = new DialogFormO;
```

Применение наследования с формами

При помощи *наследования* (inheritance) создают похожие формы на основе имеющихся, при этом новой форме передаются все члены, элементы управления, меню и код родительской формы (инструмент Inheritance Picker предоставляет графический интерфейс для использования наследования). Такую форму часто применяют в качестве основы для разработки формы с дополнительной функциональностью. Кроме того, новые формы можно объявлять на основе имеющихся, непосредственно в коде. Более подробно об этом — в главе 4.

▶ **Создание производных форм с помощью Inheritance Picker**

1. Выберите в меню Project команду Add Inherited Form — откроется диалоговое окно Add New Item.
2. В левой панели этого окна щелкните Local Project Items, а в правой панели — Inherited Form. Введите в поле Name имя формы и щелкните Open, чтобы вызвать Inheritance Picker.
3. Inheritance Picker покажет все формы проекта. Если форма, на основе которой вы хотите создать новую форму, указана в этом списке, выберите ее и щелкните OK — к проекту будет добавлена новая форма, производная от указанной.
4. Чтобы создать новую форму на основе формы из другого проекта, щелкните кнопку Browse и найдите проект, в котором хранится требуемая форма. Выберите имя DLL-файла с нужной формой и щелкните Open, чтобы вернуться в диалоговое окно Inheritance Picker.

   Теперь выбранный проект отображается в окне инструмента. Укажите нужную форму и щелкните OK — к проекту будет добавлена производная форма.

Примечание Для использования Inheritance Picker родительская форма должна располагаться в текущем проекте либо в компилированном EXE- или DLL-файле.

▶ **Как объявить производную форму прямо в коде**

1. Выберите в меню Project команду Add Windows Form — к проекту добавится новый класс формы.
2. Откройте исходный текст новой формы в редакторе кода и откорретируйте объявление класса (если используется C#) либо (если это проект Visual Basic)

добавьте ключевое слово *Inherits,* чтобы объявить эту форму как производную, например, так:

Visual Basic .NET

```
' Этот пример предполагает, что базовым является класс
' формы MainForm, расположенной в текущем проекте,
Public Class myForm
   Inherits MainForm
   ' Реализация класса опущена.

End Class
```

Visual C#

```
// Этот пример предполагает, что базовым является класс
// формы MainForm, расположенной в текущем проекте.
public class myForm : MainForm
{
   // Реализация класса опущена.
}
```

Примечание Как и в предыдущем примере, для применения наследования в проекте необходима ссылка на сборку, в которой располагается родительская форма, либо эта форма должна быть членом текущего проекта.

Назначение стартовой формы

Если приложение Windows Forms содержит несколько форм, следует одну из них назначить стартовой. При запуске приложения она загружается первой. При использовании Visual **Basic** .NET можно назначить стартовую форму, определив стартовый объект приложения в окне свойств проекта (рис. 2-1).

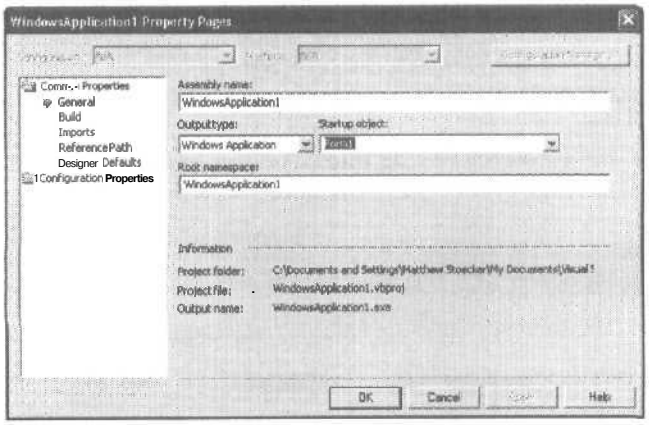


Рис. 2-1. Окно свойств проекта **Visual Basic .NET**

▶ **Как назначить стартовую форму в проекте Visual Basic .NET**

1. Щелкните имя проекта в окне Solution Explorer.
   Имя выбранного объекта подсвечивается.
2. В меню Project выберите команду Properties.
3. Укажите в списке Startup Object требуемую форму.

Назначить стартовую форму в проектах Visual C# чуть сложнее. Чтобы сделать форму стартовым объектом, объявите в ней метод *Main,* который служит точкой входа в приложение. Таким образом, размещение метода *Main* определяет стартовую форму приложения. В следующем примере наличие метода *Main* делает форму myForm стартовой формой приложения:

Visual C#

```
static void Main()
{
   Application.Run(new myForm());
}
```

Если в проекте несколько форм с методом *Main,* стартовой можно сделать любую из них. Для этого также используют окно свойств проекта (рис. 2-2).

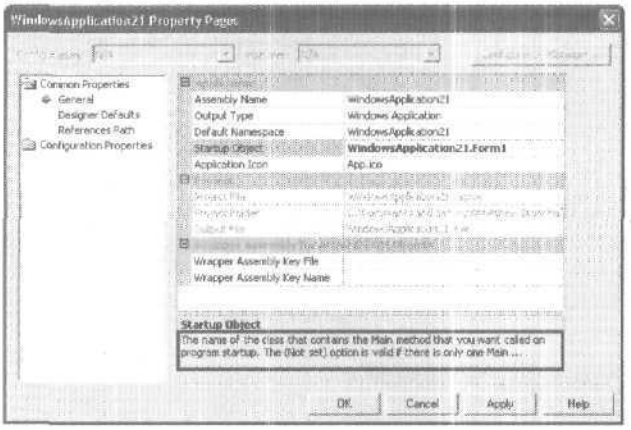


Рис. 2-2. **Окно свойств проекта Visual C#**

▶ **Как назначить стартовую форму в Visual C#**

4. Выберите в окне Solution Explorer нужный проект — его имя будет выделено подсветкой.
5. В меню Project щелкните элемент Properties.
6. Выберите из списка Startup Object требуемую форму. В ней должен быть метод *Main,* определяющий допустимый стартовый объект.

Выбор первоначального размещения формы

Свойство формы *StartPosition* позволяет определить место, в котором форма появится на экране впервые. Это свойство может принимать любое значение из перечисления *StartPosition* (таблица 2-1).

Чтобы определить первоначальное размещение формы, назначьте соответствующее значение свойству *StartPosition*.

Таблица 2-1. Значения свойства *StartPosition*

Значение	Результат
Manual	Место отображения формы определяется ее свойством *Location*
CenterScreen	Форма отображается в центре экрана
WindowsDefaultLocation	Форма отображается в месте, заданном по умолчанию в Windows
WindowsDefaultBounds	Форма отображается в месте и с размерами окна, заданными по умолчанию в Windows
CenterParent	Форма центрируется относительно своей родительской формы

Изменение внешнего вида формы

Внешний вид пользовательского интерфейса — важная составляющая приложения. Неудачный интерфейс трудно осваивать, а значит, требуется больше времени и средств на обучение работе с приложением. Внешний вид формы можно изменять, корректируя ее свойства.

У формы множество «врожденных» свойств, которые определяют ее облик. Их значения можно просматривать и изменять в дизайнере через окно свойств формы (рис. 2-3).

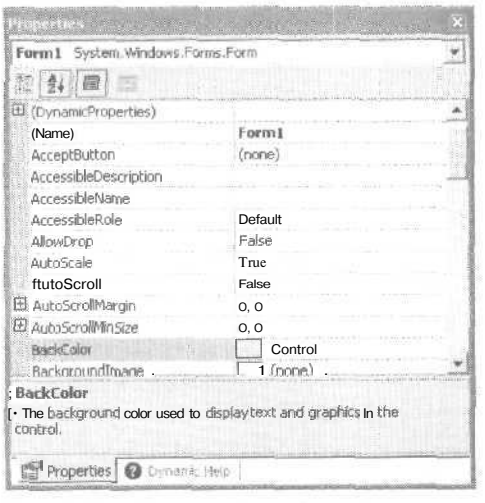


Рис. 2-3. Окно свойств формы

Некоторые свойства, такие, как *Font*, в действительности заключают в себе несколько значений, каждое из которых влияет на вид формы по-своему. Щелкнув в окне Properties знак «+» рядом с именем свойства (*Font*), вы сможете просмотреть и изменить каждое из этих значений. Для некоторых свойств, таких, как *BackColor* и *ForeColor*, значения определяют средствами специального редактора.

Свойства формы также разрешено менять в коде во время выполнения. Например, следующая команда назначает фону красный цвет:

Visual Basic .NET

```
' Предполагается, что следующий метод вызывают
' из кода формы, цвет фона которой нужно изменить.
Me.BackColor = System.Drawing.Color.Red
```

Visual C#

```
// Предполагается, что следующий метод вызывают
// из кода формы, цвет фона которой нужно изменить.
this.BackColor = System.Drawing.Color.Red
```

Для работы со свойствами принят тот же синтаксис, что и для других членов класса. Значения свойств устанавливают при помощи оператора присваивания (=) и получают, обращаясь к свойствам по имени.

Свойства BackColor, ForeColor и Text

С результатом работы этих свойств пользователи сталкиваются в первую очередь. Свойство *Text* определяет текст заголовка формы, а *BackColor* к *ForeColor* — цвета, которые используются на форме. Свойство *ForeColor* определяет цвет текста переднего плана. При размещении на форме большинство элементов управления автоматически устанавливают свое свойство *ForeColor* согласно значению одноименного свойства формы. Аналогично ведет себя свойство *BackColor (оно* определяет цвет фона) у многих элементов управления, в том числе Button и Label: при перетаскивании их в окно дизайнера это свойство автоматически устанавливается в соответствии со свойством *BackColor* формы. У остальных элементов управления, например у TextBox, это свойство определяется независимо от фона, поэтому его приходится задавать вручную.

Поскольку выбор цвета важен для дизайна пользовательского интерфейса, следует отнестись к нему очень внимательно. Возможно, во время разработки красный текст на голубом фоне выглядит забавно, но после развертывания приложения читать такой текст очень трудно. Чтобы облегчить чтение форм, следует применять высококонтрастные цветовые схемы.

Свойства Font, Cursor и BackGroundImage

Их можно считать дополнительными «рычагами» для управления пользовательским интерфейсом. Свойство *Font* позволяет указать шрифт для всех элементов управления формы. Помните, что он должен быть гармоничным и облегчать восприятие содержимого. Свойство *Cursor* задает вид указателя мыши, который он принимает, находясь над формой. Свойство *BackGroundImage* позволяет использовать изображение вместо однотонного фона. Если задано фоновое изображение, то изменение свойства *BackColor* формы отразится на значении одноименного свойства всех размещенных на ней элементов управления, но никак не повлияет на саму форму.

Прозрачность

Варьируя прозрачность формы, удается получать замечательные эффекты. Прозрачностью формы управляет свойство *Opacity*, принимающее значение из диапазона 0-1, Значение 1 означает полную непрозрачность, значение 0 — полную прозрач-

ность, а промежуточные значения — разную степень прозрачности. По умолчанию свойство *Opacity* равно 1. Свойство *Opacity* следует применять, когда требуется держать одну форму на переднем плане, держа в поле зрения ту, что расположена на заднем плане.

Обычно свойство *Opacity* элементов управления устанавливается в соответствии с одноименным свойством их формы-контейнера.

Чтобы создать прозрачную или полупрозрачную форму, присвойте свойству *Opacity* значение меньше 1, например, так:

Visual Basic .NET

```
' Эта команда делает форму полупрозрачной
MyForm.Opacity = .5
```

Visual C#

```
// Эта команда делает форму полупрозрачной
MyForm.Opacity = .5;
```

В окне Properties значение свойства *Opacity* отображается в процентах, поэтому во время разработки для этого свойства можно выбирать значение от 0 до 100%.

Применение методов формы

Методы применяются для исполнения тех или иных действий. Методы, являющиеся членами класса, выполняют действия, составляющие функциональность данного класса. Любая форма инкапсулирует базовый набор функций, унаследованный от класса *System.Windows.Forms.Form*, куда входят методы, управляющие отображением формы и доступом к ней в пользовательском окружении. Вот некоторые из них:

- *Form.Show*
- *Form.ShowDialog*
- *Form.Activate*
- *Form.Hide*
- *Form.Close*

Для вызова любого из этих методов необходима ссылка на форму, то есть в памяти должен быть заранее созданный экземпляр формы. При запуске приложение автоматически создает экземпляр стартовой формы в дополнение к тем экземплярам форм, которые создаются в коде.

Внутри класса формы можно ссылаться на текущий экземпляр этой формы посредством ключевого слова *Me* (Visual Basic) или *this* (C#). Допустим, нужно написать метод, изменяющий цвет фона формы. Поскольку этот код воздействует только на текущий экземпляр формы, задачу удается решить лишь при помощи указанного ключевого слова. Вот пример метода, модифицирующего заголовок формы:

Visual Basic .NET

```
' Эта команда модифицирует заголовок текущего экземпляра формы
Me.Text = "This is the active form"
```

Visual C#

```
// Эта команда модифицирует заголовок текущего экземпляра формы
this.Text = "This is the active form";
```

Методы Show и ShowDialog

Пользователи работают только с видимыми формами. Чтобы сделать форму видимой, вызовите метод *Form.Show*, который загружает экземпляр класса формы в память, отображает его на экране и передает ему фокус ввода, при этом свойство *Visible* автоматически устанавливается в *true*. Если экземпляр формы уже загружен, но пока не видим (например, если его свойство *Visible* установлено в *false),* вызов метода *Form.Show* даст тот же результат, что и установка свойства *Visible* в *true*.

Метод *Form.ShowDialog* выполняет те же действия, что и *Form.Show,* но делает окно формы модальным. Это означает, что другим формам приложения не удастся получить фокус, пока не закрыта форма, показанная при помощи метода *Form.ShowDialog*. Сделав окно формы модальным, вы заставите пользователя выполнить некоторое действие на этой форме, и только после этого он сможет продолжить работу с приложением. Например, метод *Form.ShowDialog* позволяет оповестить пользователя о том, что дискета недоступна. Ниже показан пример кода, демонстрирующий применение методов *Form.Show* и *Form. Show Dialog*:

Visual Basic .NET

```
' Этот пример предполагает наличие класса формы DialogForm
Dim myForm as New DialogForm()
' Отобразить форму как обычное окно
myForm.Show()
' Отобразить форму как модальное окно
myForm.ShowDialog()
```

Visual C#

```
// Этот пример предполагает наличие класса формы DialogForm
DialogForm myForm = new DialogForm();
// Отобразить форму а обычном окне
myForm.Show();
// Отобразить форму в модальном окне
myForm. ShowDialog();
```

Метод Activate

Если форма видима, но у нее нет фокуса, можно передать ей фокус, вызвав метод *Form.Activate* из формы активного приложения. При этом форма активируется и получает фокус ввода.

Если вызвать этот метод в окне приложения, у которого в текущий момент нет активных форм, окно этого приложения на панели задач начинает мигать. В любом случае активировать разрешено только видимые форме. Если вызвать этот метод из невидимой формы, он просто вернет управление.

Visual Basic .NET

```
myForm.Activate()
```

Visual C#

```
myForm.Activate();
```

Метод Hide

Чтобы сделать форму невидимой, вызовите метод *Form.Hide*. Форма остается в памяти, но она невидима, пока не будет вызван метод *Form.Show* или свойство *Visible* этой формы не будет установлено в *true*. Метод *Form.Hide* устанавливает свойство *Visible* в *false* (в принципе, аналогичный эффект достигается при установке этого свойства напрямую).

Visual Basic .NET

```
myForm.Hide()
```

Visual C#

```
myForm.Hide();
```

Метод Close

Чтобы закрыть ненужную форму и удалить ее из памяти, вызовите метод *Form. Close*. Этот метод закрывает все удерживаемые формой ресурсы и помечает ее как мусор, подлежащий сбору. После вызова метода *Form.Close* сделать форму видимой, вызвав метод *Form.Show*, не удастся, поскольку ресурсы этой формы уже освобождены. Вызов *Form.Close* на стартовой форме приложения завершает приложение.

Visual Basic .NET

```
myForm.Close()
```

Visual C#

```
myForm.Close();
```

Применение событий формы

События (events) отражают различные события, происшедшие с программой. Когда происходит нечто важное для приложения, оно *генерирует* соответствующее событие, которое *обрабатывают* другие компоненты приложения. Все описанные ранее методы генерируют при вызове одно или несколько событий. Разработчик может написать код, который позволит приложению реагировать на события, исполняя код (то есть обрабатывать это событие). Подробно события и их обработчики рассматриваются в следующей главе, а сейчас самое время познакомить вас с созданием обработчиков для основных событий и их применением в программах.

Любые элементы управления и формы генерируют множество разнообразных событий. Например, при вызове метода *Form.Hide* форма генерирует события *Deactivate* и *VisibleChanged*. Если разработчик хочет заставить приложение выполнять некоторые действия в ответ на событие, он пишет *обработчик события* (event handler) — метод, исполняемый при генерации события. Например, в обработчик события *Deactivate* можно вставить код, проверяющий заполнение обязательных полей формы.

Обработчик для события, генерируемого формой, создают средствами интерфейса Visual Studio .NET. Впрочем, этого достигают и напрямую, редактируя код, но такой способ сложнее, мы рассмотрим его в следующей главе. Процедура создания обработчиков событий в проектах, использующих Visual Basic .NET и в Visual C#, несколько отличается.

▶ **Создание обработчика события в проекте Visual Basic .NET**

1. Вызовите **Code** Editor и выберите в списке Class Name (он расположен вверху окна редактора кода) элемент Base Class Events.

Примечание Если создается обработчик события элемента управления, вместо Base Class Events нужно выбрать имя соответствующего элемента управления.

2. В раскрывающемся меню Method Name выберите событие, для которого хотите написать обработчик.
   В результате к коду добавляется шаблон обработчика события, в который можно вставить код, исполняемый при генерации данного события. Списки Class Name и Method Name показаны на рис. 2-4.

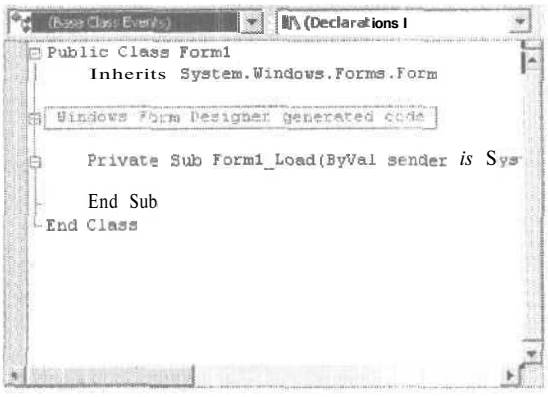


Рис. 2-4. **Создание обработчика событий в проекте Visual Basic .NET**

▶ **Создание обработчика события в проекте Visual C#**

1. В режиме Design щелкните форму или элемент управления, для которого хотите создать обработчик события.
2. Щелкните кнопку Events в окне Properties, чтобы отобразить список доступных событий.
3. Дважды щелкните событие, для которого нужно написать обработчик,

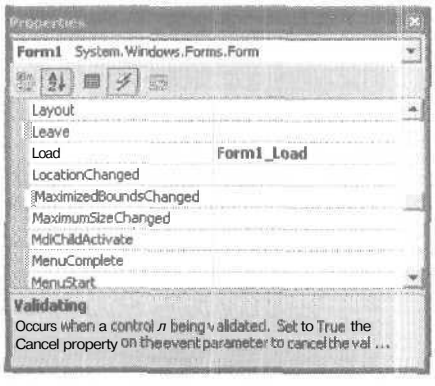


Рис. 2-5. **Создание обработчика события в проекте Visual C#**

В результате в редакторе кода откроется новый шаблон обработчика событий, после чего к обработчику можно добавить собственный код. Он будет выполняться при генерации данного события. Список событий в окне Properties показан на рис. 2-5. Обратите внимание на кнопку Events (на ней изображена молния или искра).

События жизненного цикла формы

На протяжении жизненного цикла формы генерируется ряд событий. В этом разделе мы рассмотрим те события, которые генерируются во время создания, функционирования и уничтожения формы:

- *Load*
- *Activated*
- *VisibleChanged*
- *Deactivated*
- *Closing*
- *Closed*

Хотя этот список далеко не полон, все события, которые чаще всего генерируются на протяжении жизненного цикла формы, в нем указаны.

Событие Load

Это событие генерируется, когда экземпляр формы впервые загружается в программу, то есть при первом вызове метода *Form.Show* или *Form.ShowDialog для* экземпляра формы. Рассмотрим пример:

Visual Basic .NET

```
Dim rnyForm as New Form()
myForm.ShowO              ' Здесь срабатывает событие Load
myForm.Hide()             ' Форма стала невидимой
myForm.Show()             ' Событие Load больше не срабатывает
myForm.Close()            ' Эта команда закрывает и удаляет форму...
myForm.Show()             ' ...а эта - генерирует исключение, поскольку
                          ' объект myForm уже недоступен
```

Visual C#

```
Form myForm = new Form();
myForm.Show();            // Здесь срабатывает событие Load
myForm.Hide();            // Форма стала невидимой
myForm.Show();            // Событие Load больше не срабатывает
myForm.CloseO;            // Эта команда закрывает и удаляет форму...
myForm.Show();            // ...а эта - генерирует исключение, поскольку
                          // объект myForm уже недоступен
```

Этот пример демонстрирует обстоятельства генерации события *Load*. Заметьте, что такое событие генерируется в жизненном цикле отдельной формы единственный раз. Если существует несколько экземпляров некоторой формы, то событие *Load* создается для каждого из них. Обычно в обработчике события *Load* инициализируют переменные формы и подготавливают форму к работе.

События Activated и Deactivated

Событие *Activated* многократно срабатывает в течение жизни формы. Оно генерируется, когда форма получает фокус. Так, это происходит при вызове методов *Form.Show, Form.ShowDialog* и *Form.Activate*, а также при активации формы. Обработчик события *Activated* применяют для автоматической передачи фокуса определенному элементу управления формы, а также для изменения цвета активного элемента управления, чтобы сделать его заметным пользователю.

Напротив, событие *Deactivate* генерируется, когда форма теряет фокус. Это происходит из-за взаимодействия пользователя с интерфейсом либо при вызове методов *Form.Hide* или *Form.Close* (однако метод *Form.Close* генерирует это событие, только если закрываемая форма является активной). Это событие применяют для проверки данных, введенных пользователем.

События *Activated* и *Deactivated* генерируются только при перемещении фокуса в пределах приложения: если переключиться на другое приложение и обратно, ни одно из них не сработает.

Событие VisibleChanged

Как следует из имени этого события, оно генерируется при изменении свойства *Visible* формы, то есть всякий раз, когда форма становится видимой или невидимой. Это событие генерируют методы *Form.Show, Form.ShowDialog, Form.Hide*, и *Form.Close*.

Событие Closing

Это событие генерируется, когда текущая форма начинает закрываться (но еще не закрыта), например при вызове метода *Form.Close* или щелчке кнопки Close. Это событие позволяет проверить, все ли действия, обязательные для данной формы, выполнены, например, заполнены ли все обязательные поля. В зависимости от результатов проверки, можно прервать процесс закрытия и оставить форму открытой. Для этого к обработчику этого события добавляют следующий код:

Visual Basic .NET

```
CancelEventArgs.Cancel = True
```

Visual C#

```
CancelEventArgs.Cancel = true;
```

Событие Closed

Это событие генерируется после закрытия формы. Подобно событию *Closing*, это происходит при вызове метода *Form.Close* или когда пользователь закрывает форму вручную. Событие *Closed* генерируется вслед за событием *Closing* после исполнения его обработчиков. Событие *Closed* позволяет исполнить любой код для очистки после закрытия формы.

Резюме

- Форма — это главный элемент интерфейса программ Windows Forms. Ее следует конструировать так, чтобы пользовательский интерфейс получился полным, гармоничным и привлекательным. Формы разрешается добавлять как во время разработки, так и выполнения. Наследование позволяет создавать формы со сходным оформлением и набором элементов управления, для этого применяют графический интерфейс среды разработки.
- У форм есть свойства, управляющие их внешним видом. Изменяя их, вы меняете внешний вид формы и (как правило) размещенных на ней элементов управления. Вот некоторые из этих свойств:
  - *BackColor*
  - *ForeColor*
  - *Text*
  - *Font*
  - *Cursor*
  - *BackGroundImage*
  - *Opacity*
- Для управления жизненным циклом и отображением форм предназначены следующие методы:
  - *Form.Show*
  - *Form.ShowDialog*
  - *Form.Activate*
  - *Form.Hide*
  - *Form.Close*
- Все перечисленные методы так или иначе изменяют состояние интерфейса и генерируют различные события:
  - *Load*
  - *Activated*
  - *VisibleChanged*
  - *Deactivated*
  - *Closing*
  - *Closed*
- Можно создавать специальные методы, называемые обработчиками событий. Эти методы выполняются при генерации соответствующих событий. Обработчик события вызывается каждый раз, когда генерируется обрабатываемое им событие.

Занятие 3. Применение элементов управления и компонентов

Наряду с формами пользовательский интерфейс составляют элементы управления (или, как их еще называют, элементы управления Windows Forms). Это графические инструменты, которые формируют либо дополняют функциональность приложения. Их перетаскивают на форму с инструментальной панели Toolbox. Некоторые из них, например *Button* и *TextBox*, служат для получения ввода от пользователя и выполняют базовые задачи по взаимодействию с пользователем. Кроме того, предусмотрены специализированные компоненты, реализующие более сложные операции по взаимодействию с другими частями приложения. Компоненты похожи на элементы управления: это тоже готовые блоки кода, инкапсулирующие определенную функциональность. Основное отличие между ними в том, что элементы управления видимы, а компоненты — нет (по крайней мере в период выполнения). На этом занятии вы узнаете, как применять элементы управления и компоненты при создании пользовательского интерфейса.

Изучив материал **этого занятия,** вы сможете:

- ✓ рассказать о роли элементов управления в приложении;
- ✓ объяснить различие между элементами управления и компонентами;
- ✓ задавать порядок передачи фокуса по клавише Tab для элементов управления;
- ✓ описать элементы управления, у которых могут быть вложенные элементы управления, и рассказать, как их применяют;
- ✓ рассказать, что такое *стыковка* (docking) и *фиксация* (anchoring), и объяснить, как их применяют с элементами управления;
- ✓ динамически добавлять элементы управления к форме;
- ✓ описать процесс добавления элементов управления на панель инструментов;
- ✓ создавать обработчики событий элементов управления;
- ✓ описать компоненты — провайдеры дополнительных свойств и рассказать, как их применяют.

Продолжительность занятия - около 45 минут

Работа с элементами управления

Инструментальная панель Toolbox в Visual Studio содержит множество готовых элементов управления для разработки собственных решений. Большинство реальных функций пользовательского интерфейса реализовано соответствующими элементами управления, размещенными на формах. Возьмем для примера элемент управления, представляющий кнопку (Button). Его размещают на форме и, как правило, снабжают заголовком, поясняющим его назначение. Когда пользователь щелкает кнопку, вызывается обработчик ее события *click,* исполняющий некоторый код. Элементы управления Label (надпись) и PictureBox (изображение) предназначены для отображения данных, а TextBox (текстовое поле) и ListBox (список) способны как отображать данные, так и принимать их от пользователя.

Для размещения элементов управления на форме применяют инструмент пол названием дизайнер, который отображает графическое представление формы, близкое к ее виду во время выполнения. Графический интерфейс дизайнера позволяет перетаскивать мышью необходимые элементы управления с инструментальной панели на поверхность формы. При размещении элемента управления на форме его код автоматически добавляется к приложению. Для изменения положения элементов управления, размешенных на форме, достаточно перетащить их в нужное место. Кроме того, можно изменять размеры большинства элементов управления, перетаскивая их границы. Поскольку в одной книге невозможно подробно описать все элементы управления и их функции, рекомендуем вам самостоятельно изучить элементы управления, расположенные на панели инструментов, чтобы подробнее узнать, как они работают.

▶ **Как добавить элемент управления *к* приложению**

1. **Щелкните** нужный элемент управления на панели инструментов.
2. Щелкните форму в том месте, куда его хотите поместить. Не отпуская левой кнопки мыши, установите требуемый размер элемента управления, перетащив его границы.

   Есть и другие варианты: можно просто перетащить элемент управления на форму либо дважды щелкнуть его значок на инструментальной панели — в результате на форму добавится экземпляр элемента управления, положение и размеры которого заданы по умолчанию (если это допустимо).
3. Откорректируйте размер и положение элемента управления мышью или клавишами управления курсором.

Все свойства выбранного элемента управления отображаются в окне Properties, где их можно изменять. Во время разработки некоторые свойства получают значения по умолчанию, которые разрешается корректировать во время выполнения. Для размещения элементов управления на форме используют интерфейс дизайнера. В принципе, все эти действия можно выполнить и в коде, но преимущество дизайнера в том, что он позволяет быстро задать первоначальные значения для элементов управления без затраты драгоценного времени на программирование всех свойств элементов управления формы вручную,

▶ **Редактирование свойств элемента управления во время разработки**

1. **Щелкните** элемент управления правой кнопкой и выберите из контекстного меню Properties. Возможны другие варианты: щелкните нужный элемент управления левой кнопкой и нажмите F4 либо выберите команду Properties из меню View.
2. Введите нужные значения в поля редактируемых свойств.

▶ **Как одновременно отредактировать свойства для нескольких элементов управления во время разработки**

1. Выделите нужные элементы управления на форме. Для этого щелкните форму и, не отпуская кнопки мыши, обведите их либо нажмите клавишу Ctrl и, не отпуская ее, щелкните по очереди необходимые элементы управления.

   Общие для всех выделенных элементов управления свойства отобразятся Б окне Properties.
2. Введите нужные значения в поля редактируемых свойств.

Панель Toolbox также содержит компоненты, которые, как и элементы управления, представляют собой предварительно скомпилированные блоки, инкапсулиру-

ющие определенную функциональность; их можно включать в собственные приложения. Главное отличие компонентов и элементов управления в том, что компоненты не видны в пользовательском интерфейсе приложения (например компонент *Timer,* генерирующий события через заданный интервал времени). У компонентов нет графического интерфейса. Поэтому те, что размещаются на форме при помощи дизайнера, попадают не на поверхность формы, а в *область компонентов* (component tray). Она находится внизу окна дизайнера и предоставляет графический интерфейс для манипулирования свойствами компонентов, не имеющих собственного интерфейса. Компоненты добавляют к приложению так же, как элементы управления, а для редактирования их свойств во время разработки применяется окно Properties.

Установка порядка передачи фокуса при нажатии клавиши Tab

Клавишей Tab пользователи приложения передают фокус ввода от одного элемента управления к другому. Чтобы задать последовательность перехода фокуса между элементами управления, следует определить *порядок передачи фокуса при нажатии клавиши Tab* (tab order) с помощью свойства *TabIndex.* Изменяя его, вы меняете порядок передачи фокуса: элементы управления с меньшим значением свойства *Tablndex* первыми получают фокус и передают его элементу управления с большим значением этого свойства. Если у нескольких элементов управления значение свойства *Tablndex* совпадает, первым получает фокус элемент управления переднего плана или ближайший к нему элемент. Чтобы переместить элемент управления, размещенный на форме, на передний или на задний план, щелкните его правой кнопкой и выберите соответственно Bring To Front или Send To Back.

В Visual Studio предусмотрен графический инструмент для установки порядка передачи фокуса. Чтобы вызывать его, выберите из меню View элемент Tab Order - каждый элемент в окне дизайнера будет помечен прямоугольником с числом внутри. Чтобы задать порядок передачи фокуса, достаточно щелкнуть элементы управления в том порядке, в котором они должны получать фокус.

Примечание Некоторые элементы управления, такие, как PictureBox, не могут получать фокус, поэтому у них нет свойства *Tablndex.*

▶ **Как установить порядок передачи фокуса при помощи свойства Tablndex**

1. В окне дизайнера выбирайте по очереди все элементы управления, способные получать фокус.
2. Для выбранного элемента управления установите в окне Properties требуемое значение свойства *Tablndex* — фокус будет передаваться между элементами управления в порядке возрастания значений *Tablndex.*

▶ **Как установить порядок передачи фокуса через графический интерфейс Visual Studio**

1. В меню View выберите элемент Tab Order — все элементы управления будут отмечены прямоугольниками с цифрами, отражающими текущий порядок передачи фокуса.
2. Щелкните элементы управления в желаемом порядке передачи фокуса.
3. Еще раз щелкните в меню View элемент Tab Order, чтобы возобновить редактирование формы.

Вложенные элементы управления и контейнеры

Некоторые элементы управления, называемые *контейнерами* (container controls), способны содержать другие элементы управления. К контейнерам относятся Panel, GroupBox и TabControl, форму тоже можно считать контейнером. Контейнеры позволяют упорядочить элементы управления, размещенные на формах. Например, логически связанные переключатели можно объединить средствами GroupBox. Контейнеры объединяют элементы управления в логические группы, которыми удается манипулировать программно. Применение контейнеров облегчает создание интерфейса, выдержанного в едином стиле, в котором проще ориентироваться пользователю,

Примечание Контейнер содержит другие элементы управления, но не зависит от них. Этим он отличается от пользовательских элементов управления, объединяющих несколько элементов управления в группу, все члены которой зависят друг от друга. О пользовательских элементах управления рассказано в главе 7.

Изменение свойств контейнера отражается на элементах управления, которые он содержит. Например, назначив свойства *Enabled* элемента управления GroupBox в *false*, вы автоматически сделаете недоступными все вложенные в него элементы управления. Свойства контейнера, определяющие внешний вид, такие, как *BackColor*, *ForeColor*, *Visible* и *Font*, также действуют на вложенные элементы управления. Это упрощает оформление фрагментов пользовательского интерфейса в едином стиле. Свойства *Anchor* и *Dock* вложенных элементов управления также зависят от свойств контейнера, но о них мы поговорим в следующем разделе.

Примечание При изменении свойств, определяющих облик контейнера, автоматически изменяются соответствующие свойства его вложенных элементов. Тем не менее всегда можно переопределить любое из свойств вложенного элемента управления вручную. Однако если контейнер деактивирован, активировать его вложенные элементы управления можно, только активировав сам контейнер.

Применение элементов управления Group Box и Panel

Элементы управления GroupBox и Panel очень похожи: оба позволяют объединять другие элементы управления в логические и физические группы. Можно сказать, что эти элементы физически делят форму на отдельные «подформы». Изменение свойств Panel или GroupBox действует на все вложенные в них элементы управления. Во время разработки элементы управления, вложенные в Panel или GroupBox, можно перемещать все вместе, а во время выполнения — одновременно деактивировать их, установив свойство *Enabled* контейнера в *false*.

У GroupBox есть заголовок, позволяющий аннотировать группу вложенных в него элементов управления, текст заголовка определяется свойством *Text*. У Panel заголовка нет, зато он поддерживает прокручивание содержимого. Чтобы включить полосы прокрутки для этого элемента, установите его свойство *AutoScroll* в *true*.

Применение элемента управления TabControl

TabControl позволяет упорядочивать элементы управления путем их размещения на вкладках с ярлычками, напоминающих карточки библиотечного каталога или разделители записной книжки. TabControl состоит из нескольких элементов управле-

ния TabPage, на которых, в свою очередь, располагаются другие элементы управления. Пример применения TabControl — окно свойств приложения, где на каждой вкладке сгруппированы свойства, контролирующие определенный компонент приложения.

Свойство *TabPages* — важнейшее у элемента управления TabControl. Оно представляет собой набор элементов управления TabPage, каждый из которых обладает собственным набором свойств. Набор — это логическая группа объектов, похожая на массив; подробнее о наборах рассказано в следующей главе. Чтобы получить доступ к любому из объектов TabPage, достаточно щелкнуть вкладку, которую он представляет (рис. 2-6).

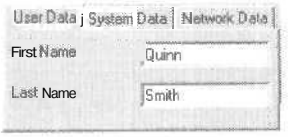


Рис. 2-6. Элементы управления TabPage в составе TabControl

Отдельные элементы управления TabPage весьма напоминают панели (Panel): они также поддерживают прокрутку и создают внутри формы-контейнера «вложенные формы», в свою очередь способные быть контейнерами для других элементов управления. Чтобы включить прокрутку содержимого для TabPage, необходимо установить его свойство *Autoscroll* в *true.*

Элементы управления TabPage добавляют к TabControl, устанавливая его свойство *TabPages.* При выборе в окне дизайнера свойства *TabPages* открывается редактор наборов, позволяющий добавлять к TabControl элементы управления TabPage и устанавливать свойства для любого из них. Чтобы разместить на добавленной вкладке необходимые элементы управления, щелкните эту вкладку в окне дизайнера. Во время выполнения пользователь сможет открывать нужные вкладки, щелкая их ярлычки.

Стыковка и фиксация элементов управления

Многие элементы управления поддерживают свойства *Anchor* и *Dock,* определяющие их поведение внутри формы или иного контейнера. Заданное свойство *Anchor* заставляет выдерживать постоянное расстояние между границами элемента управления, для которого оно определено, и границами формы. При изменении размеров формы во время выполнения размеры элемента управления, для которого задано свойство *Anchor,* автоматически изменяются таким образом, чтобы расстояние между его границами и заданными границами формы оставалось постоянным. Свойство *Dock* позволяет «пристыковать» элемент управления к одному из краев формы либо сделать так, чтоб он заполнил собой всю форму. При изменении размеров формы размеры пристыкованных к ней элементов управления редактируются автоматически.

Применение свойства Anchor

Свойство *Anchor* можно задавать для любого элемента управления во время разработки, как и любое другое свойство. Для этого выберите свойство *Anchor* в окне Properties и щелкните расположенную рядом кнопку — откроется небольшое окно (рис. 2-7), где можно выставить необходимое значение свойства.

Окно, показанное на рис. 2-7, позволяет выбрать границы, к которым прикрепляется элемент управления. По умолчанию для этого свойства принято значение

Top, Left. Оно заставляет выдерживать постоянное расстояние между левой и верхней границами формы и соответствующими границами элемента управления. Поскольку при изменении размеров формы редактируется положение ее нижнего правого угла, значение *Top, Left* в сущности заставляет сохранять положение элемента управления при корректировке размеров формы.

В этом упражнении вы добавите к форме кнопку и поэкспериментируете со свойством *Anchor*: задавая для него разные значения, вы будете наблюдать, как это влияет на положение и размеры элемента управления.

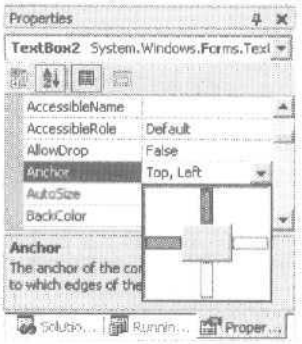


Рис. 2-7. **Установка свойства *Anchor***

▶ **Демонстрация свойства *Anchor***

1. В Visual Studio создайте новый проект Windows Forms.
2. Откройте новую форму в дизайнере и поместите на нее элементу управления Button.
3. **Щелкните** добавленную кнопку и выберите свойство *Anchor* в окне Properties.

   Откроется окно свойства *Anchor*. Заметьте, что верхняя и левая полоски в этом окне темные — это означает, что кнопка привязана к верхней и левой границе формы.
4. Измените размеры формы в дизайнере, перетащив ее правый нижний угол. Обратите внимание, что кнопка сохраняет свое положение относительно верхней и левой границ окна формы.
5. Щелкните обе затемненные полосы в окне свойства *Anchor* — теперь кнопка больше не привязана ни к одной из границ окна.
6. Попробуйте снова изменить размеры формы в окне дизайнера. Как видите, при этом положение кнопки тоже изменяется — она «плавает» на форме.

   Если свойство *Anchor y* элемента управления установлено в *None*, то при изменении размеров формы выдерживается относительное (а не абсолютное) расстояние между его границами и границами формы.
7. Установите для свойства *Anchor* значение *Top* и снова измените размеры формы. Кнопка сохраняет расстояние от верхней границы формы, но «плавает» относительно правой и левой границ.
8. Щелкните нижнюю полоску в окне свойства *Anchor* (установите для этого свойства значение «Top, Bottom»).
9. Измените размеры формы еще раз. Кнопка по-прежнему «плавает» относительно левой и правой границ формы, но расстояние до верхней и нижней границ формы останется постоянным, при этом размеры кнопки изменяются.

Свойство *Anchor* управляет реакцией элемента управления на редактирование размеров формы. Можно зафиксировать элемент управления на форме, прикрепив его к ее левой и верхней границам, либо позволить ему растягиваться вместе с формой, прикрепив его к противоположным границам формы. Наконец, вы можете разрешить элементу управления свободно «плавать» на форме, не прикрепляя его ни к одной из ее границ. Таким образом, с помощью свойства *Anchor* удается реализовать самое разное поведение элементов управления в ответ на изменение размеров формы.

Применение свойства Dock

Прикрепление элемента управления к границе его родительского элемента (контейнера) называется *стыковкой* (docking). Обычно родительским элементом управления назначают форму, но им может быть и другой элемент управления, например Panel и Tab, в качестве пристыкованного элемента управления можно выбрать меню, расположенное вверху формы.

Во время разработки значение свойства *Dock* задают в окне Properties. Если выбрать свойство *Dock,* откроется окно, где устанавливаются параметры стыковки для данного элемента управления (рис. 2-8).

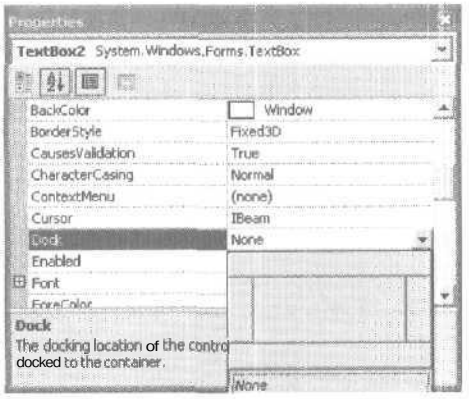


Рис. 2-8. Установка свойства *Dock*

Для назначения свойства *Dock* достаточно щелкнуть в этом окне полоску, соответствующую месту стыковки элемента управления. Например, если элемент управления следует пристыковать к верхней части формы, щелкните верхнюю горизонтальную полоску. Чтобы отменить стыковку, щелкните None. Если щелкнуть центральный квадрат в окне установки свойства *Dock,* элемент управления заполнит всю форму. Если свойство *Dock* для некоторого элемента управления определено, его свойство *Anchor* автоматически принимает значение *Top, Left.*

▶ **Как задать параметры стыковки для элемента управления**

1. В окне Properties выберите свойство *Dock* — откроется окно установки свойства *Dock.*
2. В этом окне щелкните соответствующую полоску, чтобы задать для свойства требуемое значение.

Применение наборов элементов управления

Любой элемент управления, способный выполнять функцию контейнера (в том числе и форма), поддерживает набор, состоящий из всех его вложенных элементов управления — *набор элементов управления*. У такого набора есть свойство *Count*, возвращающее число элементов набора, и свойство *Item*, возвращающее заданный элемент набора. У набора элементов управления также имеются члены-методы, позволяющие добавлять и удалять элементы из набора.

В Visual Basic .NET свойство *Item* возвращает элемент управления, заданный его параметром *Index*, определяющим индекс элемента в наборе. В Visual C# для выборки элемента из набора применяется *индексатор* (indexer), например, так:

Visual Basic .NET

```
' Этот пример предполагает наличие формы myForm
Dim aControl As Control
aControl - myForm.Controls.Item(3)
```

Visual C#

```
// Этот пример предполагает наличие формы myForm
Control aControl;
aControl = myForm.Controls[3];
```

Примечание Поскольку в Visual Basic .NET свойство *Item* задано в качестве свойства по умолчанию для набора элементов управления, при обращении к элементу набора ключевое слово *Item* можно опускать. Так, следующий пример эквивалентен предыдущему.

Visual Basic .NET

```
Dim aControl as Control
aControl = myForm.Controls(3)
```

Методы *Add* и *Remove* позволяют динамически добавлять элементы управления к набору и удалять их из него. Следующий пример демонстрирует динамическое создание надписи элемента управления Label и добавление его к набору элементов управления формы myForm.

Visual Basic .NET

```
Dim aLabel as New Label()
aLabel.Text = "This label is being added dynamically"
myForm.Controls.Add(aLabel)
```

Visual C#

```
Label aLabel - new Label();
aLabel.Text = "This label is being added dynamically";
myForrn.ControlsAdd(aLabel);
```

Метод *Remove* удаляет элементы управления из набора, а метод *RemoveAt* удобен для удаления элементов управления с заданным индексом, например:

Visual Basic .NET

```
myForm.Controls.Remove(Button1)
myForm.Controls.RemoveAt(3)
```

Visual C#

```
myForm.Controls.Remove(Button1);
myForm.Controls.RemoveAt(3);
```

Аналогичный синтаксис применяется для добавления и удаления элементов из набора элементов-контейнеров, таких, как Panel или GroupBox. Следующий пример показывает динамическое добавление элемента управления к набору TabPages элемента управления *TabControl*:

Visual Basic .NET

```
' Этот пример предполагает наличие элемента управления myTabControl
Dim aButton as New ButtonC)
' Эта команда добавляет новую кнопку на вкладку, которая входит
' в набор TabPages объекта TabControl под индексом 1.
myTabControl.TabPages(1).Controls.Add(aButton)
```

Visual C#

```
// Этот пример предполагает наличие элемента управления myTabControl
Button aButton = new Button();
// Эта команда добавляет новую кнопку на вкладку, которая входит
// в набор TabPages объекта TabControl под индексом 1.
 myTabControl.TabPages[1].Controls.Add(aButton);
```

Добавление элементов управления на панель Toolbox

Возможности разработчика не ограничены элементами управления из библиотеки базовых классов .NET Framework. Кроме них разрешено применять нестандартные элементы управления, написанные сторонними разработчиками, и создавать собственные элементы управления (об этом рассказано в главе 7). Добавив нестандартные элементы управления на инструментальную панель Toolbox, вы сделаете их доступными в среде разработки.

▶ **Как добавить элемент управления на панель Toolbox**

1. Перейдите на вкладку Toolbox, на которую хотите добавить элемент управления.
2. Щелкните правой кнопкой выбранную вкладку и выберите из контекстного меню команду Customize Toolbox — откроется окно Customize Toolbox.

   В этом окне показаны все компоненты, зарегистрированные в системе, они распределены по двум вкладкам — одна отображает .NET-компоненты, а другая — унаследованные компоненты COM.

Примечание Если панель Toolbox скрыта, выберите в меню View пункт Toolbox, чтобы открыть ее.

3. Если элемент управления, который вы хотите добавить, уже зарегистрирован в системе, укажите его в соответствующем списке и щелкните OK.

   В результате выбранный элемент управления появится на панели Toolbox,

4. Если добавляемый элемент управления находится на дискете или в другом месте, щелкните кнопку Browse и откройте каталог с файлом элемента управления.
5. Выберите этот файл и щелкните (Ж — элемент управления будет добавлен на панель Toolbox. Заметьте, что на панель Toolbox добавляются только элементы управления ActiveX, компоненты COM и .NET-сборки.

Создание обработчиков событий элементов управления

События представляют происшествия в ходе исполнения программы. Все элементы управления способны генерировать разнообразные события, отражающие ге или иные обстоятельства взаимодействия с пользователем, Рассмотрим знакомый всем пример — щелчок мыши. Когда он происходит, приложение генерирует событие *Button.Click*, ищет методы, обрабатывающие это событие и, если таковые обнаружатся, исполняет их. Такие методы называются *обработчиками события* (event handlers).

Обработчики событий создаются для исполнения некоторых действий в ответ на действия пользователя. Для каждого элемента управления задано событие по умолчанию, в качестве которого выбрано то, что генерируется этим элементом управления чаще всего. Например, событие по умолчанию для элемента управления Button — *Click,* для Checkbox — *CheckChanged.* Создать обработчик события по умолчанию проще простого — для этого достаточно дважды щелкнуть элемент управления в окне дизайнера.

▶ **Создание обработчика события по умолчанию для элемента управления**

1. Дважды щелкните нужный элемент управления в окне дизайнера — откроется окно кода с шаблоном обработчика события по умолчанию.
2. Дополните этот шаблон нужным кодом.

Помимо события по умолчанию, элементы управления способны генерировать множество других полезных событий. Например, событие *MouseOver* позволяет изменять текст, отображаемый элементом управления, когда над ним находится указатель мыши; события *Validate* и *Validating* обеспечивают поддержку проверки введенных пользователем данных (см. занятие 5) — и это далеко не все. Целый ряд других событий расширяет возможности пользовательского интерфейса и представляет пользователям разнообразную информацию. Как правило, по названию события легко догадаться, в каких случаях событие генерируется.

Обработчики событий элементов управления пишут так же, как обработчики событий формы.

▶ **Создание обработчика события в проекте Visual Basic .NET**

1. Вызовите окно Code Editor и выберите имя нужного элемента управления в списке Class Name, расположенном вверху окна редактора кода.
2. В списке Method Name выберите событие, для которого хотите написать обработчик.

   В результате в код добавляется шаблон обработчика события, который можно дополнить собственным кодом. Этот код будет выполняться всякий раз, когда элемент управления сгенерирует событие, для которого написан обработчик.

▶ **Создание обработчика события в проекте Visual C#**

1. В режиме Design щелкните элемент управления, для которого хотите создать обработчик события.
2. В окне Properties щелкните кнопку Events — откроется список доступных событий.
3. Найдите нужное событие и дважды щелкните его.

   В результате откроется редактор кода с шаблоном обработчика для указанного события, который можно дополнить собственным кодом. Этот код будет выполняться всякий раз, когда элемент управления сгенерирует событие, для которого написан этот обработчик.
4. Кроме того, можно назначить для обработки события готовый метод, выбрав его из списка, расположенного справа от списка событий.

Использование провайдеров дополнительных свойств

В этом разделе вы познакомитесь с *провайдерами дополнительных свойств* (extender providers). Это специализированные компоненты, расширяющие набор свойств элементов управления. Рассмотрим для примера компонент ToolTipProvider. После добавления экземпляра этого компонента все остальные элементы управления, размещенные на этой форме, получают новое свойство, которое разрешается просматривать и устанавливать в окне Properties. Там оно отображается под именем *ToolTip on n*, где *n* — имя экземпляра ToolTipProvider. Если во время выполнения навести указатель мыши на какой-нибудь элемент управления, около него в желтом прямоугольнике отображается значение свойства *ToolTip*.

Провайдеры дополнительных свойств обычно применяют для представления пользователям дополнительной информации во время выполнения, например, ToolTipProvider отображает всплывающие подсказки. К провайдерам дополнительных свойств также относятся элементы управления HelpProvider и ErrorProvider.

▶ **Применение провайдеров дополнительных свойств в проектах**

1. **Поместите** на форму соответствующий компонент-провайдер — он появится в области компонентов.
2. В окне Properties установите необходимые значения для свойств, поддерживаемых этим провайдером.

В действительности свойства, предоставляемые компонентами-провайдерами, содержатся в этих компонентах, а не в элементах управления, чьи возможности они расширяют, поэтому такие свойства не являются истинными свойствами компонентов. А значит, к таким свойствам нельзя обращаться во время выполнения, как к обычным свойствам. Провайдеры дополнительных свойств поддерживают специальные методы, которые позволяют получать доступ предоставляемым ими свойствам. Согласно принятым правилам, эти методы всегда называются *Getn* и *Setn*, где *n* — имя предоставляемого свойства. Так, у ToolTipProvider есть методы *GetToolTip* и *SetToolTip,* вызывая которые, можно получать и динамически изменять текст всплывающей подсказки элемента управления. Оба метода принимают в качестве аргумента ссылку на элемент управления, а методу *Set* дополнительно требуется новое значение свойства *ToolTip*.

Для получения доступа к дополнительным свойствам во время выполнения применяют соответствующий метод *Get,* которому передают ссылку на элемент управления.

Visual Basic .NET

```
' Этот пример демонстрирует получение текста подсказки
' (значения свойства ToolTip) для кнопки Button1
Dim myToolTip as String
myToolTip = ToolTip1.GetToolTip(Button1)
```

Visual C#

```
// Этот пример демонстрирует получение текста подсказки
// (значения свойства ToolTip) для кнопки Button1
string myToolTip;
myToolTip = toolTip1.GetToolTip(button1);
```

Для установки значений дополнительных свойств во время выполнения применяют соответствующий метод *Set,* которому передают ссылку на элемент управления и новое значение свойства.

Visual Basic .NET

```
' Этот пример демонстрирует установку текста
' всплывающей подсказки для кнопки Button1
ToolTip1.SetToolTip(Button1, "Click this button for help")
```

Visual C#

```
// Этот пример демонстрирует установку текста
// всплывающей подсказки для кнопки Button1
toolTip1.SetToolTip(button1, "Click this button for help");
```

Резюме

- Порядок передачи фокуса при нажатии клавиши Tab для элементов управления, размещенных на форме, устанавливают двумя способами: через значение свойства *Tab Index* либо выбрав в меню View элемент Tab Order и щелкнув элементы управления в желаемом порядке передачи фокуса.
- Некоторые элементы управления могут быть контейнерами для других элементов управления, позволяя объединять элементы управления формы в логические и физические группы. К ним, например, относятся;
  - *Panel*
  - *GroupBox*
  - *TabPage* (в составе *TabControl*)
- Свойства *Dock* и *Anchor* позволяют реализовать автоматическое изменение размеров и расположения элементов управления на форме. Установив свойство *Dock*, вы пристыкуете элемент управления к границе формы. Свойство *Anchor* задает для элемента управления «плавающее» либо фиксированное размещение, а также разрешает или запрещает менять его размеры при редактировании размеров формы.
- Для динамического добавления элементов управления во время выполнения служит набор элементов управления формы. Чтобы добавить элемент управле-

ния к форме, следует объявить его, создать его экземпляр и добавить созданный экземпляр к набору элементов управления формы.

- На панель Toolbox разрешается добавлять дополнительные элементы управления. Для этого следует щелкнуть правой кнопкой нужную секцию панели Toolbox и выбрать необходимый элемент управления из списка либо найти DLL-файл с нужным элементом управления.
- Обработчики событий элементов управления создаются так же, как обработчики событий формы. Обработчик вызывается каждый раз, когда генерируется обрабатываемое ими событие.
- Компоненты-провайдеры предоставляют дополнительные свойства времени разработки элементам управления, размещенным на форме. Эти свойства применяют для представления пользователю дополнительной информации, например подсказок и справки.

Занятие 4. Меню

Через меню пользователям открывается доступ к важным функциям и инструментальным средствам приложения. В правильно спроектированном и организованном меню без труда удается найти любые необходимые функции, что делает работу с приложением удобной.

Изучив **материал этого** занятия, вы сможете:

- ✓ рассказать о значении меню в конструировании интерфейса;
- ✓ описать процесс создания меню с использованием компонента *MainMenu*;
- ✓ создать контекстное меню при помощи компонента *ContextMenu;*
- ✓ активировать и деактивировать элементы меню;
- ✓ задавать быстрые клавиши для доступа к меню;
- ✓ снабжать элементы меню флажками и переключателями;
- ✓ делать элементы меню невидимыми;
- ✓ динамически добавлять к меню новые элементы;
- ✓ динамически клонировать меню.

Продолжительность занятия - около 30 мин.

Меню — это простой и привычный для пользователей интерфейс, открывающий доступ к командам и функциям верхнего уровня. Рационально сконструированное меню упорядочивает функциональность приложения, что облегчает его изучение и применение. Напротив, к неудачно сконструированному меню пользователи будут обращаться лишь в случае крайней необходимости.

При разработке меню необходимо учитывать логику приложения. Элементы меню следует группировать в соответствии с функциональностью, которую они представляют. Использование быстрых клавиш для доступа к элементам меню дополнительно облегчает работу с приложением.

Конструирование меню во время разработки

Главное меню приложения создают во время разработки при помощи компонента MainMenu. Этот компонент содержит набор элементов управления MenuItem, формирующих графическое представление элементов меню во время выполнения, и управляет этим набором. Таким образом, компонент MainMenu позволяет легко и быстро создать меню для формы.

Применение компонента MainMenu

Компонент MainMenu позволяет выполнять следующие действия во время разработки:

- создавать новые меню;
- добавлять к существующим меню новые элементы;
- изменять свойства меню и отдельных элементов меню через окно Properties;
- создавать обработчики *Click* и других событий элементов меню.

Все, что нужно сделать для создания меню, — это поместить на форму компонент MainMenu, который появится в области компонентов. На форме же появится полоска с прямоугольным полем, содержащим надпись «Type Here». Введите в поле

текст, чтобы создать новый элемент меню. После добавления первого элемента снизу и справа от него появляются новые поля. Вложенные элементы меню создаются аналогично: достаточно ввести имя элемента в поле, расположенное справа от родительского элемента. Вид меню в период разработки почти не отличается от такового во время выполнения. На рис. 2-9 показано создание меню при помощи компонента MainMenu.

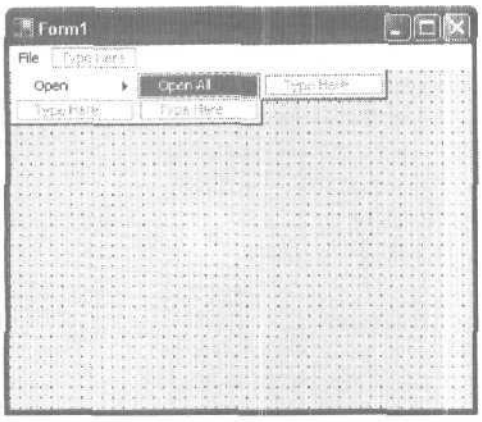


Рис. 2-9. Создание меню при помощи компонента MainMenu

При добавлении к меню нового элемента дизайнер создает экземпляр объекта *MenuItem*. У каждого объекта *MenuItem* имеются собственные свойства и члены, которые устанавливают в окне Properties. Свойство *Text* определяет текст данного элемента меню, отображаемый во время выполнения. В это свойство автоматически заносится текст, введенный в поля компонента MainMenu во время разработки. Свойство *Name* определяет имя объекта меню, которое будет использоваться для ссылок на него в коде. При создании объекта это свойство автоматически получает значение по умолчанию, которое разрешается изменять впоследствии.

▶ **Создание главного меню во время разработки**

1. Поместите на форму компонент MainMenu, перетащив его с Toolbox либо дважды щелкнув его значок на инструментальной панели — MainMenu появится в области компонентов.
2. В окне дизайнера введите в поле, расположенное на полосе меню формы, имя первого элемента меню. Компонент MainMenu откроет новые поля — укажите в них имена дополнительных элементов меню. Продолжайте, пока не создадите все элементы меню.
3. В окне Properties задайте необходимые значения для свойств элементов меню.
4. Проверьте окно свойств формы. Убедитесь, что свойство *Menu* определяет именно то меню, которое вы только что создали. Если у формы несколько меню, отображаться будет только то, которое задано этим свойством.

Разделение элементов меню

При необходимости элементы меню отделяют друг от друга горизонтальной линией — разделителем. Разделители позволяют упорядочить меню, которое состоит из множества элементов, разбив его на логические группы (рис. 2-10).

Чтобы добавить к меню разделитель, введите в поле MainMenu дефис вместо имени — во время выполнения он превратится в разделитель.

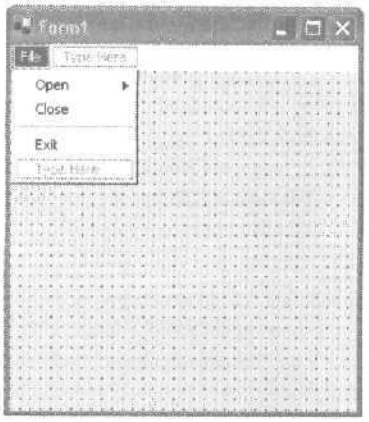


Рис. 2-10. Разделитель в меню

▶ **Создание разделителя в меню**

1. Выберите элемент меню, на место которого хотите поместить разделитель.
2. Введите в его поле дефис — во время выполнения на месте этого элемента меню будет отображаться разделитель.

Доступ к меню с клавиатуры

Доступ к элементам меню с клавиатуры осуществляют, назначая им *клавиш доступа* (access keys) и *быстрые клавиши* (shortcut keys).

Клавиши доступа

Нажимая одновременно Alt и букву, назначенную клавишей доступа, вы быстро откроете меню. Воспользовавшись клавишей доступа, можно также выбрать команду из открытого меню. Например, во многих программах Alt+F открывает меню File. Определить, какая буква назначена в качестве клавиши доступа, просто: нужная буква подчеркнута в названиях элементов меню.

Разрешается назначать одинаковые клавиши для доступа к элементам меню, если последние расположены в разных группах. Например, сочетание Alt+C применяют для доступа к командам Close из группы File и Copy из группы Edit. Однако не стоит назначать одинаковые клавиши доступа к элементам меню из одной группы, например Alt+C для команд Cut и Copy из группы Edit. Если пренебречь этой рекомендацией, клавиша доступа будет поочередно активировать элементы меню, для которых она назначена, но выбрать какой-либо из них удастся только нажатием Enter.

▶ **Как назначить клавишу доступа для элемента меню**

1. В окне дизайнера щелкните элемент меню, для которого хотите назначить клавишу доступа.
2. Введите знак «&» перед буквой, соответствующей клавише доступа.

Быстрые клавиши

Быстрые клавиши ускоряют доступ к командам меню. Эта возможность экономит время при вызове часто используемых команд. Быстрых клавиш может быть одна или несколько, например F1, Insert, Delete или Ctrl+A, Ctrl+F1, Ctrl+Shift+X. Быстрые клавиши отображаются справа от имени элемента меню, для которого они

назначены. Если для некоторого элемента меню свойство *ShowShortcut* установлено в *false,* то быстрые клавиши для него не отображаются.

▶ **Назначение быстрых клавиш**

1. Выберите элемент меню, для которого хотите назначить быстрые клавиши,
2. Щелкните свойство *Shortcut* в окне Properties.
3. Выберите из раскрывающегося списка подходящее сочетание клавиш.

События элементов меню

Процедура создания обработчика события элемента меню не отличается от таковой для других элементов управления (см. занятие 2). С элементами меню чаше всего применяют событие *Click.* Код обработчика этого события исполняется в ответ на щелчок соответствующего элемента меню, а также при нажатии назначенных ему быстрых клавиш.

Событие *Select* генерируется при выделении элемента меню подсветкой с помощью мыши или клавиш доступа. Для этого события можно написать обработчик, отображающий подробное описание команды при выборе соответствующего элемента меню.

Событие *Popup* элемента меню генерируется непосредственно перед отображением его вложенных элементов. Это событие позволяет активировать и деактивировать отдельные элементы группы команд меню перед отображением этой группы во время выполнения.

Создание контекстного меню

Контекстным называется меню, вызываемое щелчком правой кнопки мыши. Для создания контекстного меню предназначен компонент ContextMenu, редактируют его так же, как MainMenu. Компонент ContextMenu располагается наверху формы. Чтобы добавить к нему элементы меню, просто введите их имена в предоставленные поля.

Контекстное меню во многом напоминает главное меню: оно также содержит набор элементов меню и управляет этим набором. Для элементов контекстного меню можно задавать быстрые клавиши, но поддержки клавиш доступа для них не предусмотрено. Чтобы связать контекстное меню с формой или элементом управления, занесите в его свойство *ContextMenu* имя этого меню.

▶ **Создание контекстного меню**

1. Поместите на форму компонент ContextMenu, перетащив его с Toolbox или дважды щелкнув его значок на инструментальной панели — ContextMenu появится в области компонентов.
2. В окне дизайнера введите в поле компонента ContextMenu, расположенное в области меню формы, имя первого элемента. Когда появятся дополнительные поля, введите в них имена дополнительных элементов меню; продолжайте, пока не создадите все элементы меню.
3. В окне Properties задайте необходимые значения для свойств элементов меню.
4. Выберите форму или элемент управления, с которым хотите связать данное контекстное меню. В окне свойств элемента управления или формы занесите в свойство *ContextMenu* имя компонента контекстного меню. Если во время выполнения щелкнуть правой кнопкой этот элемент управления или форму, откроется назначенное контекстное меню. Одно контекстное меню разрешается связывать

с несколькими элементами управления, но у элемента управления может быть только одно контекстное меню.

Изменение меню во время выполнения

Меню можно динамически изменять в зависимости от условий времени выполнения. Например, если в данный момент отдельная команда недоступна, имеет смысл деактивировать вызывающий ее элемент. Можно представить пользователю дополнительную информацию, снабжая элементы меню флажками и переключателями. Чтобы закрыть пользователю доступ к тем или иным элементам меню, когда это нежелательно, их стоит сделать невидимыми. Кроме того, во время выполнения можно добавлять элементы к меню, а также объединять меню и клонировать их.

Активация и деактивация команд меню

У каждого элемента меню есть свойство *Enabled.* Установив его в *false,* вы деактивируете элемент меню, запрещая ему реагировать на действия пользователя, в том числе на клавиши доступа и быстрые клавиши. Деактивированные элементы меню выделены бледно-серым цветом. Следующий пример показывает, как деактивировать элемент меню во время выполнения.

Visual Basic .NET

```
MenuItem1.Enabled = False
```

Visual C#

```
menuItem1.Enabled = false;
```

Отображение элементов меню с флажками

Свойство *Checked* можно снабдить элемент меню флажком, который отображается справа от имени элемента. Флажки позволяют пометить выбранные параметры. Следующий пример демонстрирует, как пометить элемент меню флажком и снять флажок:

Visual Basic .NET

```
' Пометить элемент меню флажком
MenuItem1.Checked = True
' Снять флажок у элемента меню
MenuItem1.Checked = False
```

Visual C#

```
// Пометить элемент меню флажком
MenuItem1.Checked = true;
// Снять флажок у элемента меню
MenuItem1.Checked = false;
```

Отображение элементов меню с переключателями

Вместо флажка можно отображать переключатель. Для этого свойство *RadioCheck* элемента меню следует установить в *true,* в результате справа от имени элемента

меню отобразится не флажок, а переключатель. Если же свойство *Checked* установлено в *false,* ни фдажок, ни переключатель не отображаются. Заметьте, что переключатели часто применяют для выбора взаимоисключающих параметров, например цвета фона. Чтобы при установке одного переключателя остальные сбрасывались, вам придется самостоятельно написать соответствующий код.

Как сделать элемент меню невидимым

Чтобы сделать элемент меню невидимым, установите его свойство *Visible* в *false.* Это свойство позволяет изменять меню во время выполнения в зависимости от некоторых условий. Вот как это делается:

Visual Basic .NET

```
MenuItem1.Visible = False
```

Visual C#

```
menuItem1.Visible = false;
```

Заметьте, что элементы, сделанные во время выполнения невидимыми, удаляются из меню, а их вложенные элементы становятся недоступными.

Клонирование меню

Во время выполнения разрешается копировать (клонировать) имеющиеся элементы меню. Например, имеет смысл клонировать элемент главного меню Edit (со всеми вложенными элементами) и поместить копию в контекстное меню некоторого элемента управления. Эта задача решается с помощью метода *CloneMenu.* Этот метод создает копию заданного элемента меню со всеми его членами, к которым относятся вложенные элементы меню, свойства и обработчики событий. В результате получается клон, способный обрабатывать все события, который обрабатывал оригинал. Созданное таким образом контекстное меню достаточно просто связать с элементом управления. Следующий пример демонстрирует клонирование элемента меню во время выполнения и создание контекстного меню на основе клона:

Visual Basic .NET

```
' Этот пример предполагает наличие элемента меню fileMenuItem
' и элемента управления myButton.
' Объявить новое контекстное меню и создать его экземпляр
Dim myContextMenu as New ContextMenu()
' Клонировать fileMenuItem и занести клон з свойство myContextMenu
myContextMenu.MenuItems.Add(fileMenuItem.CloneMenu())
' Назначить контекстное меню элементу управления myButton
myButton.ContextMenu = myContextMenu
```

Visual C#

```
// Этот пример предполагает наличие элемента меню fileMenuItem
// и элемента управления myButton.
// Клонировать fileMenuItem и занести клон в свойство myContextMenu
ContextMenu myContextMenu = new ContextMenu();
```

```
// Клонируем fileMenuItem и заносим в контекстное меню элемент-клон
myContextMenu.MenuItems.Add(fileMenuItem.CloneMenu());
// Назначить контекстное меню элементу управления myButton
myButton.ContextMenu = myContextMenu;
```

Слияние меню во время выполнения

Порой во время выполнения требуется объединить содержимое нескольких меню, Для этого применяют метод *MergeMenu*, который способен объединять содержимое главного и контекстного меню, отдельные элементы меню и добавлять отдельные элементы в существующее меню.

▶ **Как объединить меню во время выполнения**

Вызовите метод *MergeMenu* из объекта меню или элемента меню, к которому нужно присоединить меню (или элемент), заданное аргументом этого метода:

Visual Basic .NET

```
MainMenu1.MergeMenu(ContextMenu1)
```

Visual C#

```
mainMenu1.MergeMenu(contexMenu1);
```

Создание элементов меню во время выполнения

Разрешается динамически добавлять к меню новые элементы во время выполнения. Например, к меню File можно добавить элементы, отображающие путь к недавно открывавшимся файлам. У новых элементов меню нет обработчиков событий, однако можно задать метод, обрабатывающий событие *Click,* передав ею конструктору элемента меню как аргумент. Этот метод должен быть процедурой *(Sub* или, в C#, *void) с* сигнатурой обработчика события. Вот примеры подходящих методов:

Visual Basic .NET

```
Public Sub ClickHandler (ByVal sender As Object. ByVal e As _
   System.EventArgs)
    ' Реализация опущена
End Sub
```

Visual C#

```
public void ClickHandler (object sender, System.EventArgs e)
{
   // Реализация опущена
}
```

▶ **Создание элементов меню во время выполнения**

1. Объявите новый элемент меню и создайте его экземпляр. При желании можно сразу же задать метод-обработчик события *Click:*

Visual Basic .NET

```
' Этот пример предполагает наличие метода myClick
' с сигнатурой обработчика события
Dim myItem as MenuItem
myItem = New MenuItem("Item 1", New EventHandler(AddressOf myClick))
```

Visual C#

```
// Этот пример предполагает наличие метода myClick
// с сигнатурой обработчика события
MenuItem myItem;
myItem = new MenuItem("Item 1", new EventHandler(myClick));
```

2. Добавьте новый элемент к набору элементов расширяемого меню.

Visual Basic .NET

```
MainMenu1.MenuItems.Add(myItem)
```

Visual C#

```
MainMenu1.MenuItems.Add(myItem);
```

Резюме

- Меню — это интерфейс, из которого легко и быстро открывается доступ к командам высокого уровня. Компонент MainMenu позволяет оперативно создать меню для приложения, а разделители, клавиши доступа и быстрые клавиши расширяют его возможности.
- Контекстное меню удобно для доступа к набору команд, который зависит от текущего контекста приложения. Контекстное меню создают средствами компонента *ContextMenu* или во время выполнения, как главное меню.
- Во время выполнения структуру меню разрешено динамически изменять, что позволяет настраивать предоставляемый пользователю интерфейс. Можно активировать, деактивировать и скрывать элементы меню, а также снабжать их флажками или переключателями. Метод *CloneMenu* позволяет динамически изменять структуру меню, создавая новые меню и объединяя существующие, а также добавляя к меню отдельные элементы.

Занятие 5. Проверка данных, вводимых пользователем

Как правило, через интерфейс пользователи передают приложениям различную информацию. Проверка вводимых данных гарантирует, что пользователю разрешается продолжить работу с программой только после ввода данных, отвечающих заданным параметрам. Предположим, что в группе полей для ввода адреса есть поле почтового индекса. Прежде чем принять введенное в это поле значение, следует удостовериться, что пользователь ввел именно пять символов, причем все пять - цифры. Проверка введенных пользователем данных уменьшает вероятность ошибки ввода и повышает устойчивость приложения.

На этом занятии вы научитесь применять события для проверки пользовательского ввода и передачи фокуса ввода другим формам и узнаете, как выполнить проверку на уровне полей (когда поля проверяются по мере их заполнения) или на уровне формы (когда все поля проверяются одновременно). Вы также научитесь задавать диапазон допустимых значений при помощи свойств элемента управления и применять компонент ErrorProvider для отображения пользователю сообщений с описанием допущенной им ошибки.

Изучив материал этого занятия, вы сможете:

- ✓ объяснить различие между проверкой на уровне формы и на уровне поля;
- ✓ передавать фокус ввода при помощи методов и событий элементов управления;
- ✓ выполнять проверку значений формы на уровне полей;
- ✓ выполнять проверку значений формы на уровне формы.

Продолжительность занятия - около 30 минут.

Разработчик вправе указать один из двух типов проверки вводимой информации: на уровне поля и на уровне формы. Проверка на уровне формы выполняется после того, как пользователь заполнит все поля формы. Предположим, что пользователь должен заполнить поля для ввода имени, адреса и номера телефона, а затем щелкнуть OK. Если задана проверка на уровне формы, все поля формы проверяются одновременно после щелчка кнопки OK.

При использовании проверки на уровне поля все поля проверяются по отдельности по мере их заполнения. Например, прежде чем пользователь перейдет к следующему полю после ввода телефонного номера, в указанном им номере проверяется код города. Применение событий элементов управления позволяет по мере ввода символов номера удостовериться, что вводимые символы являются цифрами.

Проверка на уровне поля

В ряде случаев необходимо проверять данные сразу же после их ввода, что позволяет осуществлять проверку полей по мере их заполнения. В этом разделе рассказано о том, как применять события элементов управления для проверки вводимых данных, а также задавать интервалы допустимых входных значений при помощи некоторых свойств элемента управления TextBox.

Применение свойств элемента управления TextBox

Чаще всего для приема данных от пользователя применяется элемент управления TextBox. Некоторые из его свойств позволяют ограничивать диапазон значений, вводимых в текстовое поле, например:

- *MaxLength*
- *PasswordChar*
- *Readonly*
- *MultiLine*

MaxLength

Свойство *MaxLength* ограничивает число символов, которые можно ввести в текстовое поле. Если пользователь попытается ввести больше символов, чем задано свойством *MaxLength*, текстовое поле не примет избыточные символы, а пользователь услышит звуковой сигнал. С помощью этого свойства удобно создавать поля для ввода значений фиксированной длины, например почтовых индексов.

PasswordChar

Свойство *PasswordChar* позволяет скрывать от посторонних глаз значение, вводимое во время выполнения. Например, если сделать значением свойства *PasswordChar* звездочку (*), текстовое поле будет отображать все вводимые пользователем символы как звездочки. Этот прием обычно используют для защиты паролей в окнах входа.

Для замены пароля вы можете назначить любой допустимый символ, например точку с запятой или знак «&». Независимо от назначенного символа, свойство *Text* всегда содержит то значение, которое реально ввел пользователь.

ReadOnly

Свойство *ReadOnly* определяет, разрешено ли пользователю редактировать значение текстового поля. Если это свойство установлено в *true,* пользователю не удастся изменить отображаемый в поле текст, в противном случае значение текстового поля можно редактировать, как обычно.

MultiLine

Свойство *Multiline* определяет, одна или много строк в поле. Если оно установлено в *true*, пользователь может вводить многострочный текст, завершая строки символом возврата каретки. Введенные строки сохраняются в виде строкового массива в наборе *TextBox. Lines.* Чтобы получить нужную строку, следует указать ее индекс в массиве.

Применение событий для проверки на уровне поля

Обработка событий, связанных с клавиатурой, на уровне поля позволяет немедленно проверять любые данные, вводимые пользователем. Элементы управления, способные принимать ввод с клавиатуры, генерируют следующие три события:

- *KeyDown*
- *KeyPress*
- *KeyUp*

События KeyDown и KeyUp

Нажатие и освобождение любой клавиши сопровождается генерацией события *KeyDown* и *KeyUp* соответственно. Источником событий является элемент управления, обладающий фокусом ввода. Элемент управления, генерировавший событие, передает сведения о нажатой (или отпущенной) клавише (или сочетании клавиш) через экземпляр класса *KeyEventArgs*— класса, описывающего сочетание клавиш. В сигнатуре метода, обрабатывающего событие *KeyDown* или *KeyUp,* должен быть параметр типа *KeyEventArgs.*

Чаще всего события *KeyDown* и *KeyUp* используют, чтобы определить, нажаты ли клавиши Alt, Ctrl или Shift. Результат передается обработчику соответствующего события через ссылку на экземпляр класса *KeyEventArgs.* Его свойства *Ait, Ctrl* и *Shift* возвращают значения типа Boolean, указывающие, были ли нажаты соответствующие клавиши. Значение *True* свидетельствует о том, что клавиша была нажата, а *false* — о том, что нет. Ниже показан пример обработчика события *KeyUp*,проверяющего нажатие клавиши Alt:

Visual Basic .NET

```
Private Sub TextBox1_KeyUp(ByVal sender As Object, ByVal e As _
   System.Windows.Forms.KeyEventArgs) Handles TextBox1.KeyUp
   If e.Alt = True Then
      MessageBox.Show("The ALT key still is down")
   End If
End Sub
```

Visual C#

```
private void textBox1_KeyUp(object sender,
   System.Windows.Forms.KeyEventArgs e)
{
   if (e.Alt == true)
   MessageBox.Show("The ALT key is still down");
}
```

Свойство *KeyEventArgs.KeyCode*позволяет определить, какая именно клавиша спровоцировала событие. Это свойство возвращает код нажатой или отпущенной клавиши (соответственно при событиях *KeyDown* или *KeyUp).* Ниже показан пример простого обработчика события, отображающего сообщение с кодом нажатой клавиши:

Visual Basic .NET

```
Private Sub TextBox1_KeyDown(ByVal sender As Object, ByVal e As _
   System.Windows.Forms.KeyEventArgs) Handles TextBox1.KeyDown
   MessageBox.Show(e.KeyCode.ToString())
End Sub
```

Visual C#

```
private void textBox1_KeyDown(object sender,
   System.Windows.Forms.KeyEventArgs e)
{
```

```
    MessageBox.Show(e.KeyCode.ToString());
}
```

KeyPress

Когда пользователь нажимает клавишу, которой соответствует значение ASCII, генерируется событие *KeyPress*. К этим клавишам относятся все алфавитно-цифровые клавиши (a—z, A—Z, 0—9), а также ряд специальных клавиш, таких, как Enter и Backspace. Если при нажатии клавиши или их комбинации не генерируется ASCII-символ, событие *KeyPress* также не генерируется. К таким клавишам относятся клавиши-модификаторы Ctrl и Alt, а также все функциональные клавиши.

Это событие очень удобно для перехвата нажатия клавиш и проверки соответствующих символов. При генерации события *KeyPress* обработчик получает экземпляр класса *KeyEventArgs*, свойство *KeyEventArgs.KeyCode* которого содержит ASCII-символ клавиши, нажатие которой спровоцировало это событие. Чтобы проверить, например, была ли нажата цифра, достаточно проверить свойство *KeyChar* в обработчике события *KeyPress*.

Проверка вводимых символов

Тип данных *Char* поддерживает несколько статических *[Shared (static)]* методов, удобных для проверки символов, переданных событием *KeyPress:*

- *Char.IsDigit*
- *Char.IsLetter*
- *Char.IsLetterOrDigit*
- *Char. IsPunctuation*
- *Char.IsLower*
- *Char.IsUpper*

Все они проверяют символы и возвращают булевы значения. Что проверяет каждый метод — легко догадаться по его имени. Функция *Char.IsDigit* возвращает *true,* если переданный ей символ является цифрой, и *false* в любом другом случае; *Char.IsLower* возвращает *true,* если ее аргументом является буква в нижнем регистре, и *false* в противном случае; сходным образом работают остальные методы. Вот пример применения метода *Char.IsDigit* для проверки нажатия цифр на клавиатуре:

Visual Basic .NET

```
Private Sub TextBox1_.KeyPress (ByVal sender as Object, ByVal e As _
   System.Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress
   If Char.IsDigit(e.KeyChar) - True Then
      MessageBox.Show("You pressed a number key")
   End If
End Sub
```

Visual C#

```
private void textBox1_KeyPress (object sender,
   System.Windows.Forms.KeyPressEventArgs e)
{
   if (Char.IsDigit(e.KeyChar) == true)
```

```
        MessageBox.Show("You pressed a number key");
}
```

Работа с фокусом ввода

Объект, обладающий фокусом, способен получать пользовательский ввод, осуществляемый мышью или через клавиатуру. На форме может быть несколько элементов управления, но в каждый момент времени фокус ввода только у одного из них. Элемент управления, обладающий фокусом, всегда находится на активной форме приложения.

У каждого элемента управления есть метод *Focus,* который передает фокус ввода вызвавшему его элементу управления. Метод *Focus* возвращает булево значение, свидетельствующее об успешной или неудачной передаче фокуса. Деактивированные или невидимые элементы управления не получают фокус ввода. Определить, способен ли данный элемент управления получить фокус ввода, позволяет его свойство *CanFocus*: если оно возвращает *true,* элемент управления может получить фокус, а если *false* — нет.

Visual Basic .NET

```
' Проверить, может ли TextBox1 получить фокус,
' и, если да, передать ему фокус.
If TextBox1.CanFocus = True Then
   TextBox1.Focus()
End If
```

Visual C#

```
// Проверить, может ли TextBoxl получить фокус,
// и, если да, передать ему фокус.
if (textBox1.CanFocus == true)
   textBox1.Focus();
```

События, связанные с передачей фокуса, генерируются в следующем порядке:

1. *Enter*
1. *Got Focus*
3. *Leave*
4. *Validating*
5. *Validated*
6. *LostFocus*

События *Enter* и *Leave* генерируются, когда фокус переходит к элементу управления (но еще не получен им) и покидает его. События *GotFocus* и *LostFocus* генерируются при получении и потере фокуса элементом управления. В принципе, эти события можно применять для проверки вводимых значений на уровне поля, однако события *Validating и Validated лучше* подходят для этой цели.

События Validating и Validated

Проще всего проверить вводимые данные при помощи события *Validating,* генерируемого перед потерей фокуса элементом управления. Это событие генерируется, только если у элемента управления, который получит фокус следующим, свойство *Causes Validation* установлено в *true.* Поэтому, если значение элемента управления

предполагается проверять при помощи события *Validating,* для элемента управления, который получит фокус следующим, свойство *CausesValidation* следует установить в *true.* Кроме того, использование события *Validating* требует, чтобы свойство *CausesValidation у* проверяемого элемента управления было установлено в *true.* У всех элементов управления, созданных во время разработки, свойство *CausesValidation* установлено в *true* по умолчанию, обычно исключение составляет лишь кнопка Help.

Событие *Validating* позволяет выполнять довольно сложную проверку значений элементов управления. Обработчик этого события способен, например, проверять соответствие введенного значения некоторому весьма специфическому формату или запрещать передачу фокуса другому элементу управления, пока пользователь не введет какое-либо значение.

Событие *Validating* включает экземпляр *CancelEventArgs*— класса с единственным свойством *Cancel.* Если введенное значение не отвечает заданным параметрам, проверив свойство *Cancel* в обработчике события *Validating,* можно отменить дальнейшую обработку этого события и вернуть фокус исходному элементу управления.

Событие *Validated* генерируется после успешной проверки значения элемента управления и позволяет выполнить некоторые действия в зависимости от результатов проверки.

Ниже показан пример обработчика события *Validating,* который не разрешает передать фокус следующему элементу управления, пока пользователь не введет значение в поле TextBoxl.

Visual Basic .NET

```
Private Sub TextBox1_Validating(ByVal sender As Object, ByVal e As _
   System.ComponentModel.CancelEventArgs) Handles TextBox1.Validating
   ' Проверить значение TextBoxl
   If TextBox1.Text = "" Then
      ' Если поле TextBoxl пусто, вернуть ему фокус
      e.Cancel = True
   End If
End Sub
```

Visual C#

```
private void textBox1_Validating(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   // Проверить значение TextBox1
   if (textBox1.Text == "")
      // Если поле TextBox1 пусто, зернуть ему фокус
      e.Cancel = true;
}
```

▶ **Применение события Validating для проверки текстового поля**

1. Поместите на форму текстовое поле.
2. Создайте для него обработчик события *Validating,* устанавливающий свойство *e. Cancel* в *true,* чтобы прервать проверку и вернуть фокус текстовому полю.

3. Для всех элементов управления, которые не должны генерировать событие *Validating,* установите свойство *Causes Validation* в *false.*

Проверка на уровне формы

Проверка на уровне формы позволяет одновременно проверить все поля формы. Для подобной проверки обычно применяют процедуру, которая вызывается, когда пользователь готов открыть другую форму; более совершенный способ — обработка на уровне формы события, связанного с клавиатурой.

Ниже показан пример метода, выполняющего проверку на уровне формы. По щелчку кнопки btnValidate этот метод проверяет, все ли текстовые поля формы заполнены. Если обнаружено пустое поле, метод передает ему фокус.

Visual Basic .NET

```
Private Sub btnValidate_Click(ByVal sender As System.Object, ByVal e _
   As System.EventArgs) Handles btnValidate.Click
   Dim aControl As System.Windows.Forms.Control
   ' Проверить все элементы управления формы в цикле.
   For Each aControl In Me.Controls
      ' Если этот элемент управления - текстовое поле,
      ' проверить, не пусто ли оно.
      If TypeOf aControl Is TextBox AndAlso aControl.Text = "" Then
         ' Если обнаружено пустое текстовое поле,
         ' передать ему фокус и выйти из метода.
         aControl.Focus()
         Exit Sub
      End If
   Next
End Sub
```

Visual C#

```
private void btnValidate_Click(object sender, System.EventArgs e)
   {
   // Проверить все элементы управления формы в цикле.
   foreach (System.Windows.Forms.Control aControl in this.Controls)
   {
      // Если этот элемент управления - текстовое поле,
      // проверить, не пусто ли оно.
      if (aControl is System.Windows.Forms.TextBox & aControl.Text ==
         "")
      {
         // Если обнаружено пустое текстовое поле,
         // передать ему фокус и выйти из метода.
         aControl.Focus();
         return;
      }
   }
}
```

Обработка событий клавиатуры на уровне формы

Обработка связанных с клавиатурой событий на уровне формы — более сложная методика, чем показанная только что. Централизованная обработка событий, связанных с клавиатурой, позволяет управлять вводом данных в любое поле формы. Например, можно написать метод, активирующий командные кнопки только после заполнения всех полей формы и выполняющий определенные действия в зависимости от того, какие клавиши нажимаются.

Обработку событий на уровне формы реализуют с применением событий *KeyPress*, *KeyDown* и *KeyUp*. Форма автоматически генерирует события клавиатуры, только если на ней нет активированных или видимых элементов управления, в противном случае эти события генерирует элемент управления, получающий фокус. Чтобы заставить форму автоматически генерировать события клавиатуры, следует установить ее свойство *KeyPreview* в *true* — в результате форма будет генерировать эти события прежде элемента управления, получившего фокус. Предположим, что событие *KeyPress* обрабатывается и формой, и размещенным на нем текстовым полем, а свойство *KeyPreview* формы установлено в *true*. При нажатии клавиши форма первой генерирует событие *KeyPress*, поэтому ее обработчик этого события исполняется первым, и только после его завершения будет исполнен обработчик события *KeyPress* текстового поля.

Оповещение пользователя об ошибках ввода

Если пользователь ввел в поле недопустимое значение, необходимо оповестить его об этом и дать возможность исправить ошибку. Существует много способов уведомления об ошибках ввода. Если ошибка очевидна и ее не требуется пояснять, можно ограничиться звуковым сигналом. В Visual Basic .NET для этого предназначен метод *Beep*.

Visual Basic .NET

```
' Эта команда подает звуковой сигнал
Beep()
```

Примечание В Visual C# нет встроенных методов, отвечающих за подачу звуковых сигналов.

Привлечь внимание пользователя к ошибке можно и по-другому, изменив цвет фона или текста элемента управления (при помощи его свойств *BackColor* и *ForeColor* соответственно). Например, выделить текстовое поле с недопустимым значением, задав для него красный фон через свойство *BackColor*.

Чтобы вывести более информативное описание ошибки, воспользуйтесь методом *MessageBox.Show*, отображающим небольшое модальное окно с сообщением. Поскольку это окно — модальное, пользователю не удастся просто игнорировать его и продолжить работу с программой. Вот пример вызова метода *MessageBox.Show;*

Visual Basic .NET

```
MessageBox.Show("That value is not valid for this control")
```

Visual C#

```
MessageBox.Show("That value is not valid for this control");
```

Компонент ErrorProvider

Компонент ErrorProvider предоставляет удобный способ оповещения пользователей о допущенных **ими** ошибках ввода. **Он** позволяет задать для каждого элемента управления формы текст сообщения, отображаемого при вводе недопустимого значения. **Если** для элемента управления задан текст сообщения об ошибке, то при возникновении ошибки рядом с ним появится соответствующий значок, а если навести на него указатель мыши, отобразится всплывающая подсказка с заданным сообщением. Компонент ErrorProvider расположен в секции Windows Forms панели Toolbox.

Отображение сообщений об ошибках

Метод *SetError* компонента ErrorProvider позволяет вывести сообщение об ошибке рядом с элементом управления. Этот метод принимает имя элемента управления и текст сообщения об ошибке в качестве параметров; вызывают его так:

Visual Basic .NET

```
' Этот пример предполагает наличие элемента управления nameTextBox
' и компонента ErrorProvider с именем rnyErrorProvider.
myErrorProvider.SetError(nameTextBox, "Name cannot be left blank!")
```

Visual C#

```
// Предполагается существование элемента управления nameTextBox
// и компонента ErrorProvider с именем rnyErrorProvider.
myErrorProvider.SetError(nameTextBox, "Name cannot be left blank!");
```

В результате исполнения этого кода поле nameTextBox отображает значок, а при наведении на этот элемент управления указателя мыши появляется всплывающая подсказка с заданным текстом.

Сообщение об ошибке разрешается задавать и во время проектирования Если изучить окно Properties после добавления на форму компонента ErrorProvider, нетрудно заметить, что у каждого элемента управления появилось новое свойство *Error on x,* где x — имя экземпляра ErrorProvider. Во время проектирования значение этого свойства задают через **окно** Properties, во время выполнения заданное таким образом значение отображается как сообщение об ошибке для данного элемента,

Ряд свойств компонента ErrorProvider определяет способ отображения сообщения об ошибке. Свойство *Icon* задает значок, отображаемый после элемента управления. Одна форма может содержать несколько экземпляров ErrorProvider, например один, отображающий сообщения об ошибках, а другой — предупреждения, при этом каждому экземпляру ErrorProvider разрешается назначить собственный значок. Другое свойство этого компонента — *BlinkStyle* — заставляет значок мигать, частоту мигания определяет свойство *BlinkRate.*

▶ **Применение компонента ErrorProvider при создании обработчика события, проверяющего значение элемента управления**

1. Создайте форму и добавьте к ней компонент ErrorProvider — он появится в области компонентов.
2. Установите в *true* свойство *CausesValidation* элемента управления, который должен выводить сообщения об ошибках, если это еще не сделано.

3. Добавьте к обработчику события *Validating* этого элемента управления код, проверяющий введенное в него значение. При помощи метода *SetError* установите текст сообщения, которое отображается, если при проверке введенного обнаружится ошибка. Вот пример обработчика, использующего экземпляр компонента ErrorProvider с именем myErrorProvider; этот обработчик проверяет текстовое поле pswordTextBox:

Visual Basic .NET

```
Private Sub pswordTextBox_Validating(ByVal sender as Object, _
   ByVal e As System.ComponentModel.CancelEventArgs) Handles _
   pswordTextBox.Validating
   ' Проверить введенное значение.
   If pswordTextBox.Text = "" Then
      ' Задать текст сообщения об ошибке ввода.
      myErrorProvider.SetError(pswordTextBox, _
         "Password cannot be blank!")
   Else
      ' Если введено допустимое значение, очистить текст сообщения:
      ' поскольку ошибки нет, сообщение не выводится.
      myErrorProvider.SetError(pswordTextBox, "")
   End If
End Sub
```

Visual C#

```
private void pswordTextBox_Validating(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   // Проверить введенное значение.
   if (pswordTextBox.Text == "")
      // Задать текст сообщения об ошибке ввода.
      myErrorProvider.SetError(pswordTextBox,
         "Password cannot be blank!");
   else
      // Если введено допустимое значение, очистить текст сообщения:
      // поскольку ошибки нет, сообщение не выводится.
      myErrorProvider.SetError(pswordTextBox,        "");
}
```

Резюме

- При проверке вводимых данных на уровне формы все поля формы проверяются одновременно, а при проверке на уровне поля проверка отдельных полей выполняется по мере их заполнения. Проверка на уровне поля лучше подходит для контроля значений отдельных полей.
- У элемента управления *TextBox* есть свойства времени разработки, определяющие допустимые значения текстового поля:
  - *MaxLength*
  - *PasswordChar*
  - *ReadOnly*
  - *MultiLine*
- События, связанные с клавиатурой, позволяют проверять символы, генерируемые при нажатии клавиш и их сочетаний. Эти события генерируются элементом управления, обладающим фокусом ввода и получающим ввод с клавиатуры: если свойство *KeyPreview* формы установлено в *true,* первой их генерирует форма. Вот список этих событий:
  - *KeyDown*
  - *KeyPress*
  - *KeyUp*
- Методы структуры Char удобны для проверки вводимых символов;
  - *Char.IsDigit*
  - *Char. Is Letter*
  - *Char.IsLetterOrDigit*
  - *Char. IsPunctuation*
  - *Char. IsLower*
  - *Char. Is Upper*
- Событие *Validating* применяется для проверки вводимых данных. Оно генерируется перед потерей фокуса элементом управления, только если у элемента управления, который получает фокус следующим, свойство *CausesValidation* установлено в *true.*
- При помощи компонента *ErrorProvider* элементам управления назначают текст сообщения и значок для оповещения об ошибке ввода во время выполнения. Для отображения сообщений об ошибках во время выполнения применяют метод *ErrorProvider. Set Error.*

Практикум 2. Приложение Virtual Doughnut Factory

В этом практикуме вы создадите пользовательский интерфейс для приложения Virtual Doughnut Factory на Visual Basic .NET или Visual C#. Это приложение предназначено для управления виртуальной пекарней, в которой выпекаются пончики, и магазином, где их продают. Практикум поможет вам закрепить навыки применения элементов управления, конструирования меню и реализации проверки вводимых данных. Решение задач этого практикума см. в каталоге \Labs\Ch02\Solution на прилагаемом к книге компакт-диске.

Прежде всего

Для выполнения этого практикума достаточно освоить материал глав 1 и 2.

Продолжительность практикума - около 60 мин.

Упражнение 1. Создание пользовательского интерфейса

В этом упражнении вы создадите пользовательский интерфейс, добавив к форме необходимые элементы управления.

▶ **Создание нового проекта**

1. Выберите язык, с которым вы будете работать — Visual Basic .NET или Visual C#. Создайте новый проект на основе шаблона Windows Application и назовите его **Joe's Doughnut Factory.**
2. В результате будет создан проект с одной формой. В окне Solution Explorer щелкните Form1 — окно Properties покажет свойства этой формы. Поменяйте значение свойства File Name на **frmMain.**
3. Задайте для свойств формы значения, указанные в таблице 2-2.

Таблица 2-2. Свойства первой формы проекта

Объект	Свойство	Значение
Form1	*Name*	**frmMain**
	Text	**The Doughnut Factory**
	Size	**480, 400 (приблизительно)**
Joe's Doughnut Factory (только для Visual Basic .NET)	*Startup Object*	**frmMain**

4. Если вы используете Visual C#, найдите метод *static void Main()* и внесите в него следующие изменения:

Visual C#

```
static void Main()
{
   Application.Run(new frmMain());
}
```

5. Добавьте к форме следующие элементы управления и задайте для них свойства, указанные в таблице 2-3.

Таблица 2-3. Элементы управления формы frmMain и их свойства

Имя элемента управления	Тип	Свойство	Значение
Label1	Label	*Name*	lblTitle
		Text	Current Inventory
		Font.Size	14
		Font. Bold	True
Label2	Label	*Name*	lblRaised
		Text	Raised
Label3	Label	*Name*	lblCake
		Text	Cake
Label4	Label	*Name*	lblFilled
		Text	Filled
Label5	Label	*Name*	lblGlazedRaised
		Text	Glazed
TextBox1	Text Box	*Name*	txtGlazedRaised
		Text	Оставьте пустым
		Readonly	True
Label6	Label	*Name*	lblSugarRaised
		Text	Sugar
TextBox2	TextBox	*Name*	txtSugarRaised
		Text	Оставьте пустым
		Readonly	True
Label7	Label	*Name*	lblChocolateRaised
		Text	Chocolate
TextBox3	TextBox	*Name*	txtChocolateRaised
		Text	Оставьте пустым
		ReadOnly	True
Label8	Label	*Name*	lblPlainCake
		Text	Plain
TextBox4	TextBox	*Name*	txtPlainCake
		Text	Оставьте пустым
		Readonly	True
Label9	Label	*Name*	lblChocolateCake
		Text	Chocolate
TextBox5	TextBox	*Name*	txtChocolateCake
		Text	Оставьте пустым
		ReadOnly	True

Таблица 2-3. (*продолжение*)

Имя элемента управления	Тип	Свойство	Значение
Label10	Label	*Name*	lblSugarCake
		Text	Sugar
TextBox6	Text Box	*Name*	txtSugarCake
		Text	Оставьте пустым
		Readonly	True
Label И	Label	*Name*	lblLemonFilled
		Text	Lemon
Text Box 7	TextBox	*Name*	txt Lemon Filled
		Text	Оставьте пустым
		Readonly	True
Label12	Label	*Name*	lblGrapeFilled
		Text	Grape
TextBox8	TextBox	*Name*	txtGrapeFilled
		Text	Оставьте пустым
		ReadOnly	True
Label13	Label	*Name*	lblCustardFilled
		Text	Custard
TextBox9	TextBox	*Name*	txtCustard Filled
		Text	Оставьте пустым
		Readonly	True
Label14	Label	*Name*	lblSale
		Text	Current Sale
Label15	Label	*Name*	lblQuantity
		Text	Quantity
Label16	Label	*Name*	lblType
		Text	Type
Label17	Label	*Name*	lblPrice
		Text	Price
Text Box 10	Text Box	*Name*	txtQuantity
		Text	0
ComboBox1	Combo Box	*Name*	cmbType
		Text	Оставьте пустым
Text Box 11	TextBox	*Name*	txt Price
		Text	Оставьте пустым
		ReadOnly	True
ListBox1	List Box	*Name*	lstSale
Label18	Label	*Name*	lblTotal
		Text	Total

Таблица 2-3. (*окончание*)

Имя элемента управления	Тип	Свойство	Значение
Text Box 12	TextBox	*Name*	txtTotal
		Text	Оставьте пустым
		ReadOnly	True
Button 1	Button	*Name*	btnAddToSale
		Text	Add To Sale
Button2	Button	*Name*	btn Remove Item
		Text	Remove Item
Button3	Button	*Name*	btnCheckOut
		Text	Check Out

6. В окне Properties добавьте к набору элемента управления cmbType следующие строки: **Raised-Glazed**, **Raised-Sugar**, **Raised-Chocolate**, **Cake-Plain**, **Cake-Chocolate**, **Cake-Sugar**, **Filled-Lemon**, **Filled-Grape**, **Filled-Custard**.

Примерный вид готовой формы показан на рис. 2-11.

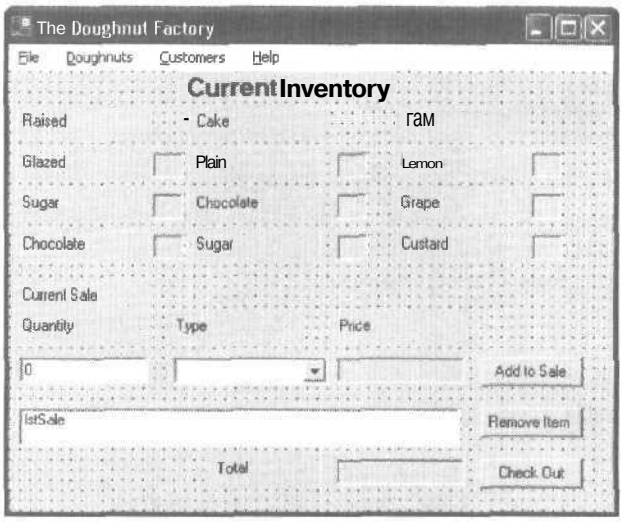


Рис. 2-11. Пользовательский интерфейс с добавленными элементами управления

Упражнение 2. Создание меню

Сейчас вы создадите меню приложения при помощи компонента MainMenu и напишете обработчик события *Click* для элемента меню Exit.

▶ Добавление меню к проекту

1. Дважды щелкните на панели Toolbox компонент MamMenu — этот компонент отобразится в области компонентов, а на месте меню на форме появится прямоугольное поле.
2. Введите в прямоугольное поле строку **&File**.

К меню добавится элемент File, а снизу и справа от него появятся пустые прямоугольные поля. Задайте клавишу доступа для элемента меню при помощи знака «&». В окне Properties назовите созданный элемент меню **mnuFile**.

3. Повторяя пункт 2, создайте элементы меню, указанные в таблице 2-4. Вложенные элементы меню указаны в таблице с отступом, они располагаются ниже соответствующих родительских элементов.

Таблица 2-4. Элементы меню

Элемент меню	Имя
&File	mnuFile
E&xit	mnuExit
& Doughnuts	mnuDoughnuts
&Make	mnuMake
&Remove Stale	mnuRemoveStale
&Customers	mnuCustomers
&Add a Customer	mnuAddaCustomer
&View Customers	mnuViewCustomers
&Help	mnuHelp
&About	mnuAbout
&Contents	mnuContents

▶ **Создание обработчика события Click для элемента меню Exit**

1. В окне дизайнера дважды щелкните элемент меню Exit — откроется окно кода с шаблоном обработчика события *mnuExit_Click*.
2. Вставьте в шаблон вызов метода *Close* формы:

Visual Basic .NET

```
Private Sub mnuExit_Click(ByVal sender As SystemObject, ByVal e As _
   System.EventArgs) Handles mneExit.Click
   Me.Close()
End Sub
```

Visual C#

```
private void mnuExit_Click(object sender, System.EventArgs e)
{
   this.Close();
}
```

3. Сохраните приложение и протестируйте его.

Упражнение 3. Создание обработчиков, проверяющих вводимые данные

Сейчас вы напишете простые обработчики, проверяющие значение текстового поля txtQuantity, Для этого воспользуйтесь компонентом Error-

Provider. В следующем практикуме вы добавите к форме дополнительные процедуры, проверяющие вводимые данные.

▶ Создание обработчиков, проверяющих вводимые данные, в проектах Visual Basic .NET

1. Дважды щелкните на панели Toolbox компонент *ErrorProvider* — он появится в области компонентов.
2. Щелкните правой кнопкой окно дизайнера и выберите из контекстного меню View Code, чтобы открыть редактор кода. В редакторе кода укажите в раскрывающемся списка слева txtQuantity, а в списке справа — *KeyPress*.

   В редакторе кода появится шаблон обработчика события *txtQuantity_KeyPress*.
3. Добавьте к обработчику код, отображающий сообщение об ошибке при вводе в это поле любых символов, кроме цифр.

Visual Basic .NET

```
Private Sub txtQuantity_KeyPress(ByVal sender As Object, ByVal e _
   As System.Windows.Forms.KeyPressEventArgs) _
   Handles txtQuantity.KeyPress
   If Char.IsDigit(e.KeyChar) = False Then
         ErrorProvider1.SetError(txtQuantity, _
         "Please enter a numeric value")
   Else
      ErrorProvider1.SetError(txtQuantity, "")
   End If
End Sub
```

4. В раскрывающемся списке слева выберите *txtQuantity*, а в меню справа — событие *Validating;* редактор кода откроет шаблон обработчика соответствующего события.
5. Добавьте к обработчику код, проверяющий наличие значения в текстовом поле txtQuantity, а также допустимость введенного значения:

Visual Basic .NET

```
Private Sub txtQuantity_Validating(ByVal sender As System.Object, _
   ByVal e As System.ComponentModel.CancelEventArgs) Handles _
   txtQuantity.Validating
   ' Проверить, пусто ли поле.
   If txtQuantity.Text = "" Then
      ErrorProvider1.SetError(txtQuantity, _
      "Please enter a quantity")
      e.Cancel = True
      ' Проверить, допустимо ли введенное значение.
   ElseIf ErrorProvider1.GetError(txtQuantity) <> "" Then
      e.Cancel = True
   Else
      ErrorProvider1.SetError(txtQuantity, "")
   End If
End Sub
```

▶ Создание обработчиков, проверяющих вводимые данные, в проектах Visual C#

1. Дважды щелкните на панели Toolbox компонент *ErrorProvider* — он появится в области компонентов.
2. В окне дизайнера выберите поле txtQuantity; в окне Properties щелкните кнопку Events и дважды щелкните событие *KeyPress* — в редакторе кода откроется шаблон обработчика события *txtQuantity_KeyPress*.
3. Добавьте к обработчику код, отображающий сообщение об ошибке при вводе в поле любых символов, кроме цифр.

Visual C#

```
private void txtQuantity_KeyPress(object sender,
   System.Windows.Forms.KeyPressEventArgs e)
{
   if (Char.IsDigit(e.KeyChar) == false)
      errorProvider1.SetError(txtQuantity,
      "Please enter a numeric value");
  else
      errorProvider1.SetError(txtQuantity, "");
}
```

4. В Solution Explorer щелкните правой кнопкой форму frmMain и выберите из контекстного меню команду View Designer. Выберите поле txtQuantity, в окне Properties щелкните кнопку Events, затем дважды щелкните событие *Validating* — редактор кода откроет шаблон обработчика соответствующего события.
5. Добавьте к обработчику код, проверяющий наличие в текстовом поле txtQuantity значение, а также допустимость введенного значения:

Visual C#

```
private void txtQuantity_Validating(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   // Проверить, пусто ли поле.
   if (txtQuantity.Text == "")
   {
      errorProvider1.SetError(txtQuantity,
         "Please enter a quantity");
      e.Cancel = true;
   }
   // Проверить, допустимо ли введенное значение.
   else if (errorProvider1.GetError(txtQuantity) != "")
      e.Cancel = true;
   else
      errorProvider1.SetError(txtQuantity, "");
}
```

Закрепление материала

Приведенные ниже вопросы помогут вам лучше усвоить основные темы данной главы. Если вы не сумеете ответить на вопрос, повторите материал соответствующего занятия. Ответы для самопроверки — в приложении «Вопросы и ответы» в конце книги.

1. Крупный банк заказал вам приложение, которое должно быть легким в освоении, бесшовно интегрироваться с Microsoft Office XP и иметь интерфейс, дизайн которого соответствует стилю банка (для воспитания у сотрудников банка корпоративной культуры). Как достичь поставленных целей? Покажите два способа создания интерфейса,

2. Вы создаете приложение, несколько форм которого имеют идентичный набор элементов управления. Как быстрее всего создать такие формы?

3. Каждый раз, когда форма получает фокус, должно отображаться приглашение на ввод значения. Опишите наилучшую стратегию для реализации подобной функциональности.

4. Что такое провайдеры дополнительных свойств и для чего они нужны?

5. Как задать порядок передачи фокуса между элементами управления на форме по нажатию клавиши Tab? Приведите два способа решения этой задачи,

6. Для каких целей лучше использовать контекстное меню?

7. Объясните, что такое «проверка на уровне формы» и «проверка на уровне поля».

ГЛАВА 3

Типы и члены

Занятие 1. Применение типов данных 100
Занятие 2. Константы, перечисления, массивы и наборы 110
Занятие 3. Реализация свойств 123
Занятие 4. Реализация событий и делегатов 132

В этой главе

В этой главе рассказывается о применении различных типов данных, констант, перечислений, массивов и наборов, а также рассматривается реализация свойств и событий.

Прежде всего

Для выполнения занятий этой главы достаточно знать материал предыдущих глав.

Занятие 1. Применение типов данных

Инфраструктура .NET Framework поддерживает целую систему элементарных типов, предназначенных для представления данных, в том числе типы для представления целых чисел, чисел с плавающей точкой, а также булевых, символьных и строковых значений. Эта система обеспечивает контроль типов, разрешая выполнять неявное преобразование типов, только если оно не приводит к потере данных. В ситуациях, чреватых потерей данных, необходимо явное преобразование типов. Типы данных .NET обладают встроенной функциональностью, позволяющей решать задачи, связанные с преобразованием типов и ряд других.

Изучив материал этого занятия, вы сможете:

- ✓ описать типы данных, поддерживаемые .NET Framework;
- ✓ выполнять явное и неявное преобразование типов;
- ✓ описать базовые возможности, поддерживаемые типами .NET Framework;
- ✓ описать способы манипулирования строками при помощи методов класса *String*.

Продолжительность занятия - около 30 минут.

Типы данных применяются для представления и хранения данных в приложении. Инфраструктура .NET Framework предоставляет богатый набор типов, позволяющих хранить значения, манипулировать ими, а также передавать их от одного члена приложения другому. Все .NET-совместимые языки являются *строго типизированными,* то есть не разрешают произвольно заменять объекты одного типа объектами других типов. При вызове методам и свойствам необходимо передавать параметры соответствующего типа, в противном случае вызов заканчивается неудачей. При необходимости можно получить из одного типа другой, воспользовавшись явным или неявным преобразованием. Это возможно, поскольку все типы .NET Framework происходят от одного — *Object,* являющегося базовым для всех классов и структур.

Типы данных .NET Framework

Типы данных в .NET применяются для хранения различных данных. Все они относятся к типам значения, их можно разбить на следующие категории: целочисленные типы, типы значений с плавающей точкой, типы булевых и символьных значений. В этом разделе мы поговорим о них, а также обсудим два встроенных ссылочных типа, без которых не обходится ни одно приложение, — *String* и *Object.*

Целочисленные типы

.NET Framework поддерживает целый набор целочисленных типов. Вместе с соответствующими им типами из языков Visual Basic .NET и Visual C# они перечислены в таблице 3-1.

Переменным целочисленного типа можно присваивать десятичные или шестнадцатеричные значения. Литералы, обозначающие шестнадцатеричные числа, следует предварять символами *&H* (в Visual Basic .NET) либо *0x* (в Visual C#), например:

Visual Basic .NET

```
Dim myInteger as Integer
myInteger = &H32EF
```

Visual C#

```
int myInteger;
myInteger = 0x32EF;
```

Таблица 3-1. Целочисленные типы .NET Framework

.NET	Visual C#	Visual Basic .NET	Описание	Диапазон значений
System.Byte	*Byte*	*Byte*	8-разрядное целое число без знака	0—255
System.Int16	*Short*	*Short*	16-разрядное целое число со знаком	-32768-32767
System. Int32	*int*	*Integer*	32-разрядное целое число со знаком	-2^{31}—2^{31}—1
System.Int64	*long*	*Long*	64-разрядное целое число со знаком	-2^{63}-2^{63}— 1
System. SByte	*sbyte*	нет	8-разрядное целое число со знаком	-128-127
System. UInt16	*ushort*	нет	16-разрядное целое число без знака	0-65535
System. UInt32	*uint*	нет	32-разрядное целое число без знака	0—2^{32}—1
System. UInt64	*ulong*	нет	64-разрядное целое число без знака	0—2^{64}—1

Типы значений с плавающей точкой

Для представления дробных чисел применяют три типа с плавающей точкой (таблица 3-2).

Таблица 3-2. Типы .NET Framework, представляющие значения с плавающей точкой

.NET Framework	Visual C#	Visual Basic .NET	Описание	Точность	Примерный диапазон значений
System. Single	*float*	*Single*	32-разрядное число с плавающей точкой	7 значащих цифр	от +/—$1{,}4\times10^{-45}$ до +/—$3{,}4\times10^{38}$
System.Double	*double*	*Double*	64-разрядное число с плавающей точкой	15-16 значащих цифр	от +/—$5{,}0\times10^{-324}$ до +/—$1{,}7\times10^{308}$
System. Decimal	*decimal*	*Decimal*	128-разрядное число с плавающей точкой	28 значащих цифр	от +/—$1{,}0\times10^{-28}$ до +/—$7{,}9\times10^{28}$

Тип *System.Single* подходит для вычислений с плавающей точкой, не требующих высокой точности, поскольку он хранит значения с точностью до седьмого знака. Тип *System.Double* обеспечивает более высокую точность и позволяет оперировать большими числами. Тип *System.Decimal* разработан специально для финансовых расчетов и представляет значения со сверхвысокой точностью. Диапазон значений этого типа существенно уже по сравнению с *System.Double*, зато он намного точнее: позволяет хранить значения с точностью до 28 знака.

Нечисловые типы

В этом разделе пойдет речь о четырех типах, не представляющих числа: *System.Boolean*, *System.Char*, *System.String* и *System.Object*.

System.Boolean

Переменные, объявленные с типом *System.Boolean*, могут принимать значение «истина» *(true)* либо «ложь» *(false)*. В Visual Basic .NET ему соответствует тип *Boolean*, принимающий значения *True* или *False*, а в Visual C# — тип *bool* (со значениями *true* и *false*).

System.Char

Тип *System.Char* представляет отдельные символы в 16-разрядной кодировке Unicode. В Visual Basic .NET ему соответствует тип *Char*, а в Visual C# — *char*. Переменной этого типа можно присвоить значение в виде литерала, состоящего из символа, заключенного в одинарные (Visual C#) или в двойные (в Visual Basic .NET) кавычки с суффиксом *«c»*:

Visual Basic .NET

```
Dim rnyChar As Char
myChar = "W"c
```

Visual C#

```
char myChar;
myChar = 'W';
```

Переменную этого типа можно инициализировать также числовым значением, представляющим код символа Unicode. В Visual Basic .NET это делается с помощью функции *Chr* или *ChrW*, а в Visual C# код символа задают четырехзначным шестнадцатеричным числом с префиксом «*\u*», заключенным в одинарные кавычки:

Visual Basic .NET

```
Dim myChar As Char
myChar = Chr(521)
```

Visual C#

```
char myChar;
rnyChar = '\u01fe';
```

System.String

Тип *System.String*— ссылочный, представляющий набор значений типа *System.Char*. Говоря повседневным языком, тип *String* способен представлять слова, абзацы, разделы — любые строки символов. В Visual Basic .NET ему соответствует тип *String*, а в Visual C# — *string*. Переменной этого типа можно присваивать значение в виде литерала, состоящего из строки, заключенной в двойные кавычки, например, так:

Visual Basic .NET

```
Dim myString As String
myString = "This is a String! Wow!"
```

Visual C#

```
string myString;
myString = "This is a String! Wow!";
```

У класса *String* много встроенных функций, о которых мы поговорим далее.

System.Object

Тип *Object* — предок всех типов .NET Framework. Любой тип, независимо от того, является ли он типом значения или ссылочным, происходит от *System.Object.* В Visual Basic .NET ему соответствует тип *Object,* а в Visual C# — *object.* В переменную этого типа можно записать любой объект или значение, например, так:

Visual Basic .NET

```
Dim myObject As Object
myObject = 543
myObject = New System.Windows.Forms.Form()
```

Visual C#

```
object myObject;
myObject = 543;
myObject = new System.Windows.Forms.Form();
```

Чтобы получить доступ к встроенной функциональности объекта некоторого типа, сохраненного в переменной типа *Object,* необходимо явно привести ее к исходному типу.

Преобразование типов

Время от времени требуется преобразовывать одни типы в другие. Подобное преобразование можно выполнить *неявно,* то есть автоматически, или *явно,* когда программист самостоятельно запрашивает преобразование типов.

Неявное преобразование

Неявное преобразование типов выполняется автоматически, если только оно не ведет к потере данных, например:

Visual Basic .NET

```
Dim anInteger As Integer = 100
Dim aLong As Long
' Поскольку тип long способен принять любое
' значение типа integer, потери данных не будет.
aLong = anInteger
```

Visual C#

```
int AnInteger = 100;
long aLong;
// Поскольку тип long способен принять любое
// значение типа integer, потери данных не будет.
aLong = anInteger;
```

Если один тип разрешается неявно преобразован в другой, его всегда можно использовать вместо второго типа, скажем, при вызове методов, причем для этого не нужен никакой особый синтаксис:

Visual Basic .NET

```
' Предполагается, что методу TakesADouble требуется параметр типа Double.
Dim I As Integer = 100
' Перед вызовом переменная I неявно
' преобразуется в Double и передается методу,
TakesADouble(I)
```

Visual C#

```
// Предполагается, что методу TakesADouble требуется параметр типа Double,
int I = 100;
// Перед вызовом переменная I неявно
// преобразуется в Double и передается методу.
TakesAdouble(I);
```

Неявные преобразования, которые поддерживаются в Visual Basic .NET и в Visual C#, перечислены в таблице 3-3.

Таблица 3-3. Неявные преобразования типов, допустимые в Visual Basic .NET и Visual C#

Исходный тип	Целевой тип
Byte (Visual Basic .NET) *byte* (Visual C#)	*Short, Integer, Long, Single, Double, Decimal* *short, ushort, int, uint, long, ulong, float, double, decimal*
Short *short*	*Integer, Long, Single, Double, Decimal* *int, long, float, double, decimal*
Integer *int*	*Long, Single, Double, Decimal* *long, float, double, decimal*
Long *long* *float,*	*Single, Double, Decimal* *double, decimal*
Single *float*	*Double* *Double*
Char *char*	*Integer, Long, Single, Double, Decimal* *Int, uint, long, ulong, float, double, decimal*
sbyte (только в Visual C#) *ushort* (только в Visual C#)	*short, int, long, float, double, decimal* *int, uint, long, ulong, float, double, decimal*
uint (только в Visual C#) *ulong* (только в Visual C#)	*long, ulong, float, double, decimal* *float, double, decimal*

Явное преобразование

Если требуемое преобразование нельзя выполнить неявно, необходимо сделать это явно. Такое преобразование называется *приведением* (cast) типов. Для явного преобразования в Visual Basic .NET служит функция *CType*, а в Visual C# для этого применяется специальный синтаксис, например:

Visual Basic .NET

```
Dim aLong As Long = 1000
Dim anInteger As Integer
anInteger = CType(aLong, Integer)
```

Visual C#

```
long aLong = 1000;
int anInteger;
anInteger = (int)aLong;
```

Однако явное преобразование таит иногда риск ошибок. В предыдущем примере приведение типов выполняется без проблем, поскольку типы *Long* и *Integer* способны принять значение 1000. А теперь рассмотрим такой пример:

Visual Basic .NET

```
Dim anInteger As Integer = 100000
Dim aShort As Short
aShort = CType(anInteger, Short)
```

Visual C#

```
int anInteger = 100000;
short aShort;
aShort = (short)anInteger;
```

При компиляции и исполнении этого примера ошибок не возникает, но при проверке значения переменной *aShort* она окажется равной –31072. Максимально возможное значение *aShort* меньше значения переменной, которую пытаются к ней привести, преобразование в результате выполняется некорректно, и переменная получает неожиданное значение (–31072).

Примечание Поскольку явное преобразование по своей природе небезопасно, следует применить его только при крайней необходимости и предусмотреть обработку последствий некорректного преобразования.

Параметр Strict в Visual Basic .NET

В Visual Basic .NET поддерживается параметр *Option Strict*. Он включает строгую проверку типов, допускающую только преобразования типов без потери данных. Если этот параметр отключен, строгая проверка типов не выполняется, и все преобразования осуществляются неявно. По умолчанию этот параметр отключен, чтобы активизировать его, необходимо в *самом начале* кода поместить ключевое слово *Option,* например, так:

Visual Basic .NET

```
Option Strict On
```

Отключение параметра *Option Strict* возможно лишь для преемственной совместимости с прежними версиями Visual Basic. Следует писать программы с параметром *Option Strict,* установленным в *On*. Это позволит «отловить» на стадии компиляции многие ошибки, которые очень трудно отследить во время выполнения.

▶ **Преобразование типов**

1. Если преобразование выполняется без потери данных, оно осуществляется неявно, явное приведение типов в этом случае не требуется.
2. Если преобразование типов чревато потерей данных, необходимо выполнить его явно. В Visual Basic .NET для этого применяется функция *CType, а* в Visual C# — специальный синтаксис.

Применение встроенной функциональности типов данных

У всех типов данных есть встроенная функциональность, включающая, по крайней мере, следующие четыре метода:

- ***Equals*** — сравнивает два экземпляра одного типа;
- ***GetHashCode*** — возвращает хеш-код для данного типа;
- ***GetType*** — возвращает объект типа для текущего экземпляра;
- ***ToString*** — возвращает объект в виде строки, понятной человеку.

Как видите, размер типов данных очень мал. Как же они поддерживают все эти методы?

Упаковка

Секрет заключается в *упаковке* (boxing) — неявном преобразовании типов значений в ссылочные типы. Все классы и типы ведут начало от базового класса *Object*, а четыре вышеупомянутых метода являются методами класса *Object.* Поскольку любой класс происходит от класса *Object,* все классы могут быть неявно преобразованы в этот тип. При вызове этих встроенных методов типа исполняющая среда создает в стеке временную ссылку, таким образом, позволяя обращаться с данными типа значения, как со ссылочным типом.

Упаковку типов значений можно выполнять и вручную. Для этого достаточно подлежащее упаковке значение записать в переменную типа *Object*:

Visual Basic .NET

```
Dim I As Integer = 100
Dim O As Object
O=I
```

Visual C#

```
int I = 100;
object O;
O = I;
```

Обратная операция — преобразование упакованной переменной в тип значения — называется *распаковкой* (unboxing). Для распаковки переменной следует явно привести ее к требуемому типу, но так можно поступать только с объектами, которые ранее были упакованы.

Собственные методы типов данных

У типов данных также есть методы, не унаследованные ими от класса *Object.* Обычно они реализуют функциональность, специфичную для типа. В главе 2 мы познакомили вас с некоторыми типа *Char,* выполняющими сравнение значений этого

типа. Этот раздел слишком мал, чтобы привести в нем достаточно полный обзор этих методов, но мы все же рассмотрим наиболее популярные.

Метод Parse

Все типы значения поддерживают метод *Parse,* позволяющий получить из строки числовое значение. **Метод** *Parse* чрезвычайно удобен при разработке пользовательского интерфейса. Элементы управления, принимающие данные от пользователя (такие, как **TextBox),** получают их в виде строк, а метод *Parse* позволяет преобразовать эти строки в числа, если же строка не может быть преобразована в число, возникает ошибка. Во всех реализациях метод *Parse* является общим *(Shared)* или статическим *(static),* поэтому его следует вызывать из типа, а не из экземпляра объекта.

▶ **Как преобразовать строку в значение числового типа**

Вызовите метод *Parse* типа данных, в который нужно преобразовать строку, например, так:

Visual Basic .NET

```
Dim I As Integer
Dim S As String
S = "1234"
' I = 1234
I = Integer.Parse(S)
```

Visual C#

```
int I;
string S;
S = "1234";
// I = 1234
I = int.Parse(S);
```

Функции для манипулирования строками

Для многих приложений жизненно необходима возможность манипулирования строками (например, преобразования строк в формат, пригодный для отображения пользователю, извлечения подстрок из имеющейся строки и т.д.). Класс *String* поддерживает разнообразные методы для работы со строками. Подробное обсуждение всех методов класса *String* не является целью этого занятия, но самые полезные из них описаны в таблицах 3-4 и 3-5. В первой таблице указаны методы экземпляров типа *String;* возвращаемая ими строка зависит от экземпляра строкового значения, из которого они вызваны. Во второй таблице перечислены общие (статические) методы, которые необходимо вызывать из типа *String.*

Таблица 3-4. Полезные методы экземпляров класса *String* (выборочно)

Имя	Описание
String. Insert	Вставляет заданную строку в текущий экземпляр
String. Pad Left, String. Pad Right	Добавляет к текущей строке символы — соответственно слева или справа
String.Remove	Удаляет из текущей строки заданное количество символов, начиная с заданного

Таблица 3-4. ***(окончание)***

Имя	Описание
String. Replace	Заменяет в текущей строке заданный символ другим заданным символом
Siring. Split	Возвращает массив разделенных заданным символом строк, полученных путем разбиения заданной строки
String. Substring	Возвращает подстроку, полученную из текущего экземпляра
String. ToCharArray	Возвращает массив символов, составляющих текущую строку
String. ToLower, String. ToUpper	Возвращают строку, преобразованную в нижний или верхний регистр соответственно
String. TrimEnd, String. TrimStart, String. Trim	Удаляют из текущей строки последний, первый или оба этих символа соответственно

Таблица 3-5. Полезные статические методы класса *String* (выборочно)

Имя	Описание
String.Compare	Сравнивает заданные экземпляры
String.Concat	Возвращает строку, полученную конкатенацией двух или более строк
String.Format	Возвращает строку, отформатированную согласно заданному формату
String.Join	Возвращает строку, полученную конкатенацией заданного массива строк, вставляя между элементами массива заданную строку-разделитель

Резюме

- .NET Framework предоставляет систему типов данных, устойчивую к ошибкам и обеспечивающую контроль типов. В ней имеются типы, представляющие целые числа, числа с плавающей точкой, а также булевы и строковые значения.
- Преобразование типов осуществляется двумя способами:
  - неявно — их исполняющая среда выполняет автоматически и только в тех случаях, когда потеря данных исключена. Пример — преобразование типа с меньшей точностью в тип с большей точностью;
  - явно — такие преобразования явно определяются в коде. Они несут потенциальную опасность потери данных, поэтому их нужно применять с осторожностью.
- Параметр *Option Strict On,* поддерживаемый Visual Basic .NET, включает строгий контроль типов во время разработки. Это позволяет выявить ошибки, связанные с несоответствием типов, на «отлов» которых в противном случае потребовалось бы очень много времени. Это параметр следует использовать, где только возможно.
- Упаковка позволяет обращаться с типами значения, как со ссылочными типами. При распаковке выполняется преобразование упакованных ссылочных типов в типы значения.
- У .NET-типов имеется встроенная функциональность, специфичная для типа. Метод *Parse* поддерживают все типы значений, он удобен для преобразования строк в типы значения. Класс *String* предоставляет ряд удобных функций для работы со строками.

Занятие 2. Константы, перечисления, массивы и наборы

Это занятие посвящено применению пользовательских констант (тип *Constant),* перечислений (тип *Enum*) и массивов (тип *Array), а* также реализации дополнительных возможностей при работе с массивами при помощи классов *Collection.*

Изучив материал этого занятия, вы сможете:

- ✓ создавать константы и перечисления и применять их в методах;
- ✓ создавать массивы и работать с ними;
- ✓ объяснить, что такое наборы, и рассказать, как их применяют;
- ✓ перебирать элементы наборов и массивов в цикле.

Продолжительность **занятия - около 40 минут.**

При работе с объектами часто требуется упорядочивать их путем организации в различные структуры. При этом необходимо последовательно получать доступ к сходным и разнородным объектам, объединенным в группы. Для выборки нужного объекта из структуры, упорядочивающей группу объектов, требуется особый синтаксис. Массивы позволяют группировать однотипные объекты и ссылаться на них, используя вместо имени индекс в массиве. Наборы напоминают массивы, но в отличие от них позволяют группировать объекты разных типов. Константы позволяют назначать часто употребляемым в программе значениям описательные имена. При помощи перечисления удается организовать логически связанные целочисленные константы в набор, чтобы сделать код читабельнее и проще в отладке.

Константы и перечисления

Зачастую некоторые значения встречаются в коде приложения многократно. Константы позволяют присваивать таким значениям дружественные имена и применять их в коде для ссылки на эти значения. Перечисление — это определяемый пользователем набор целочисленных констант с дружественными именами. Применение констант и перечислений делает код более понятным, простым в отладке и устойчивым к ошибкам, вызванным опечатками.

Применение констант

Как уже говорилось, константы позволяют использовать дружественные имена для ссылок на часто встречающиеся в коде значения. Их объявляют при помощи ключевого слова *Const* (*const*), например:

Visual Basic .NET

```
Public Const Pi as Double = 3.14159265
```

Visual C#

```
public const double Pi = 3.14159265;
```

Константами могут быть любые внутренние или перечислимые типы, но не пользовательские типы и массивы. По определению, константы — это неизменяемые значения, которые нельзя модифицировать или переопределить после того, как

они были установлены. Определив константу, можно использовать ее в коде вместо значения, которое она представляет. Вот пример константы:

Visual Basic .NET

```
Public Const Pi as Double = 3.14159265
Public Function CalculateCircleArea(ByVal r as Double) as Double
   Return Pi * r ^2
End Function
```

Visual C#

```
public const double Pi = 3.14159265;
public double CalculateCircleArea(double r)
{
   // В C# ^- это двоичный оператор, а не оператор возведения в степень.
   return Pi * r * r;
}
```

Подобно переменным, у констант бывают разные уровни доступа. Если константа должна быть доступной всем пользователям приложения или компонента, следует объявить ее с ключевым словом *Public (public),* как в предыдущем примере. Константы, предназначенные исключительно для внутреннего употребления, необходимо объявлять с модификатором *Private (private).* Константы, объявленные с ключевым словом *Friend (internal),* доступны в пределах сборки, а те, которые определены с ключевым словом *Protected (protected),*— всем типам, производным от типа, в котором она объявлена.

Применение перечислений

Перечисления позволяют работать с наборами логически связанных констант, назначая им имена, которые легко запомнить. Например, перечисление (см. пример кода ниже), объявленное на основе констант, представляющих дни недели, позволит заменить в коде номера дней недели их названиями,

Visual Basic .NET

```
Public Enum DaysOfWeek
   Monday = 1
   Tuesday = 2
   Wednesday = 3
   Thursday = 4
   Friday = 5
   Saturday = 6
   Sunday = 7
End Enum
```

Visual C#

```
public enum DaysOfWeek
{
   Monday - 1,
```

```
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}
```

По умолчанию перечисления объявляются с типом *Integer (int),* но у них может быть и любой другой целочисленный тип (в Visual Basic .NET это *Byte, Short, Integer* или *Long, a* в Visual C# — *byte, short, int* или *long).* Чтобы создать перечисление с типом, отличным от типа по умолчанию, укажите нужный тип в объявлении перечисления:

Visual Basic .NET

```
Public Enum DaysOfWeek As Byte
' Остальной код опущен
```

Visual C#

```
public enum DaysOfWeek : byte
' Остальной код опущен
```

Не обязательно указывать значения членов перечисления явно. Если значения членов перечисления не заданы, по умолчанию им присваиваются последовательные значения, начиная с нуля:

Visual Basic .NET

```
Public Enum Numbers
zero     ' имеет значение "0"
one      ' имеет значение "1"
two      ' имеет значение "2"
End Enum
```

Visual C#

```
public enum Numbers
{
   zero, // имеет значение "0"
   one,  // имеет значение "1"
   two   // имеет значение "2"
}
```

Определив члены перечисления, можно применять их в коде. В Visual Basic .NET члены перечисления автоматически преобразуются в значения, которые они представляют. Таким образом, их допустимо использовать как константы:

Visual Basic .NET

```
' Этот пример использует перечисление Numbers из предыдущего примера.
MessageBox.Show((Numbers.two * 2).ToString()) ' Эта команда выводит
                                              ' сообщение "4".
```

Однако для использования значений членов перечисления в Visual C# необходимо явно приводить их к соответствующему целочисленному типу, например:

Visual C#

```
// Этот пример использует перечисление Numbers из предыдущего примера,
MessageBox.Show(((int)Numbers.two * 2).ToString()); // Эта команда выводит
                                                    // сообщение "4".
```

Поскольку перечисления — это пользовательские типы, разрешается создавать методы, требующие в качестве параметров члены перечисления вместо числовых значений. Замена числовых параметров членами перечислений уменьшает вероятность ошибок, связанных с опечатками. Следующий пример демонстрирует применение созданного ранее перечисления *DaysOfWeek* в качестве параметра для метода:

Visual Basic .NET

```
Public Sub ScheduleDayOff(ByVal day as DaysOfWeek)
   Select Case day
      Case DaysOfWeek.Monday
         ' Реализация опущена
      Case DaysOfWeek. Tuesday
         ' Реализация опущена
      ' Остальные операторы Case опущены
   End Case
End Sub
```

Visual C#

```
public void ScheduleDayOff(DaysOfWeek day)
{
   switch(day)
   {
      case DaysOfWeek.Monday:
         // Реализация опущена
         break;
      case DaysOfWeek.Tuesday:
         // Реализация опущена
         break;
      // Прочие варианты опущены
   }
}
```

Массивы

Массивы позволяют работать с группами однотипных значений или объектов. При помощи массивов удается объединить набор переменных в группу и обращаться к этим переменным по индексу в массиве. Можно перебирать эти переменные (все или выборочно) в цикле, проверяя или изменяя их значения, или же создавать многомерные массивы. У массивов .NET Framework имеется встроенная функциональность, достаточная для решения большинства типовых задач обработки массивов.

Объявление и инициализация массивов

Массив разрешается объявить и инициализировать одним оператором, указав тип элементов массива и их число. Нумерация элементов массивов в Visual Basic .NET и Visual C# начинается с нуля — индекс первого элемента равен нулю, далее элементы нумеруются последовательно. В Visual Basic .NET число элементов массива задают путем указания его *верхней границы*. Верхняя граница массива — это индекс его последнего элемента. В Visual C# число элементов массива задают иначе, указывая его при объявлении массива. Так, верхняя граница массива, объявленного на Visual C#, всегда на единицу меньше числа его элементов. Следующий пример демонстрирует объявление и инициализацию массива целых чисел:

Visual Basic .NET

```
' Объявить и инициализировать массив из 33 целых чисел,
'  проиндексировав его от 0 до 32.
Dim myIntegers(32) As Integer
```

Visual C#

```
// Объявить и инициализировать массив из 33 целых чисел,
// проиндексировав его от 0 до 32.
int[] myIntegers = new int[32];
```

Впрочем, объявлять и инициализировать массивы можно в два этапа. В Visual C# в одной строке разрешается объявить массив, а в другой — динамически инициализировать его, как показано ниже:

Visual C#

```
// В этой строке объявляем массив,
int[]  myIntegers;
// а в этой - записываем в него 32 элемента.
myIntegers = new int[32];
```

Visual Basic .NET также позволяет динамически инициализировать массивы, но применяемый для этого синтаксис менее гибок. Если массив объявляется без указания числа его элементов, при инициализации все элементы необходимо инициализировать. Начальные значения элементов массива заключают в фигурные скобки и разделяют запятыми, например, так:

Visual Basic .NET

```
' В этой строке объявляем массив,
Dim myIntegers() As Integer
' а в этой - записываем в него шесть членов, попутно присваивая им значения.
myIntegers = New Integer() {0, 1, 2, 3, 4, 5}
```

Во время выполнения можно изменить размер массива, переопределив его. В Visual C# для этого достаточно инициализировать массив повторно:

Visual C#

```
// Объявить и инициализировать массив.
int[] mylntegers = new int[32];
```

```
// Инициализировать массив заново.
myIntegers = new int[45];
```

Для изменения числа элементов массива в Visual Basic .NET применяют оператор *ReDim*:

Visual Basic .NET

```
Dim myIntegers(32) As Integer
' Инициализировать массив заново.
ReDim myIntegers(45) As Integer
```

В предыдущих примерах повторная инициализация массива приводила к потере всех хранимых в нем данных. В Visual **Basic** .NET имеется ключевое слово *Preserve,* которое заставляет сохранять **данные** массива при его повторной инициализации; увы, в **Visual** C# это невозможно.

Visual Basic .NET

```
' Объявить массив и записать в него четыре элемента.
Dim myIntegers() As Integer = {0,1,2,3}
' Инициализировать массив заново, сохранив значения
' элементов с индексами 0-3.
ReDim Preserve myIntegers(10)
```

Создание массивов ссылочных типов несколько отличается. После объявления и инициализации получается массив, заполненный не значениями, а пустыми (null) ссылками на объекты указанного типа. Чтобы заполнить массив реальными объектами, следует записать в каждый элемент массива ссылку на новый или существующий объект данного типа:

Visual Basic .NET

```
' Создать массив объектов Widget, поместив
' во все элементы пустые ссылки на объект Widget
Dim Widgets(10) as Widget
' Поместить в элемент Widgets[0] ссылку на новый объект Widget
Widgets(0) = New Widget()
' Присвоить элементу Widgets[1] ссылку на существующий объект Widget
Dim aWidget As New Widget()
Widgets(1) = aWidget
' Перебрать в цикле элементы массива Widgets с индексами 2-10,
' записывая в них ссылки на новые объекты Widget.
Dim Counter as Integer
For Counter = 2 to 10
   Widgets(Counter) = New Widget()
Next
```

Visual C#

```
// Создать массив объектов Widget, поместив
// во все элементы пустые ссылки на объект Widget
Widget[] Widgets = new Widget[11];
// Присвоить элементу Widgets[0] ссылку на существующий объект Widget
```

```
Widgets[0] = new Widget();
// Присваиваем Widgets[1] ссылку на существующий объект Widget
Widget aWidget = new Widget();
Widgets[1] = aWidget;
// Перебрать в цикле элементы массива Widgets с индексами 2-10,
// записывая в них ссылки на новые объекты Widget.
for (int Counter = 2; Counter < 11; Counter++)
{
   Widgets[Counter] = new Widget();
}
```

Многомерные массивы

Пока мы рассмотрели только линейные массивы, то есть массивы с единственным измерением. Кроме них .NET Framework поддерживает два типа многомерных массивов: *прямоугольные* (rectangular) и *невыровненные* (jagged).

Прямоугольные массивы

Любое измерение прямоугольного массива образовано элементами, сформированными в свою очередь из одинакового числа элементов. Так, двумерный массив можно представить как таблицу, состоящую из строк с одинаковым числом ячеек.

Прямоугольные массивы создают, указывая в объявлении дополнительные измерения. Далее показаны примеры различных способов объявления многомерных прямоугольных массивов:

Visual Basic .NET

```
' Объявить массив размером 5x3.
Dim intArrays(4, 2) As Integer
' Объявить двумерный массив и задать начальные значения для его элементов.
Dim intArrays(,) As Integer = {{1, 2, 3}, {4, 5, 6}}
' Объявить трехмерный массив и задать начальные значения для его элементов.
Dim cubeArray(, ,) As Integer = {{{7, 2}, {1, 4}}, {{3, 5}, {4, 4}}}
' Объявить многомерный массив размером 3?3?4?5?6.
Dim complexArray(2, 2, 3, 4, 5) As Integer
```

Visual C#

```
// Объявить массив размером 5x3.
int[ , ] intArrays = new int[5, 3];
// Объявить двумерный массив и задать начальные значения для его элементов.
int[ , ] intArrays = {{1, 2, 3}, {4, 5, 6}};
// Объявить трехмерный массив и задать начальные значения для его элементов.
int[ , , ] cubeArray = {{{7, 2}, {1, 4}}, {{3, 5}, {4, 4}}};
// Объявить многомерный массив размером 3x3x4x5x6.
int[ , , , , ] complexArray = new int[3, 3, 4, 5, 6];
```

Невыровненные массивы

Многомерные массивы другого типа называются невыровненными. Двумерный невыровненный массив можно представить в виде таблицы, образованной строками с

разным числом ячеек, такой, как таблица, каждая строка которой представляет семью, а число ячеек в строке равно числу членов этой семьи. Ясно, что строки в такой таблице разной длины. Для представления этой и подобных ей таблиц применяются невыровненные массивы.

На самом деле невыровненный массив — это массив массивов. Чтобы его создать, необходимо объявить массив, элементами которого являются другие массивы. Для этого потребуется несколько пар скобок, причем размер невыровненного массива указывают в первой скобке. Вот как это делается:

Visual Basic .NET

```
' Объявить массив из 3 массивов.
Dim Families(2)() As String
' Инициализировать первый массив 4 элементами, присвоив им значения.
Families(0) = New String() {"Smith", "Mom", "Dad", "Uncle Phil"}
' Инициализировать второй массив 5 элементами, присвоив им значения.
Families(1) = New String() {"Jones", "Mom", "Dad", "Suzie", _
   "Little Bobby"}
' Инициализировать третий массив 3 элементами, присвоив им значения,
Families(2) = New String() {"Williams", "Earl", "Bob"}
```

Visual C#

```
// Объявить массив из 3 массивов.
string[][] Families = new string[3][];
// Инициализировать первый массив 4 элементами, присвоив им значения.
Families[0] = new string[] {"Smith", "Mom", "Dad", "Uncle Phil"};
// Инициализировать второй массив 5 элементами, присвоив им значения.
Families[1] = new string[] {"Jones", "Mom", "Dad", "Suzie",
   "Little Bobby"};
// Инициализировать третий массив 3 элементами, присвоив им значения.
Families[2] = new string[] {"Williams", "Earl", "Bob"};
```

Наборы

Наборы предоставляют дополнительные возможности по управлению группами объектов. *Набор* (collection) — это специальный класс, объединяющий несколько объектов в группу и открывающий к ней доступ. Как и в массиве, доступ к члену набора осуществляется через его индекс, но в отличие от массива размеры набора разрешается динамически изменять, добавляя и удаляя его члены во время выполнения.

Примечание В этом разделе рассматриваются только типы наборов из пространства имен *System.Collections*. Технически возможно создать класс набора и без этих типов, применяя некоторые интерфейсы, которые обычно реализуют в типах наборов. Подробнее об интерфейсах и их реализации рассказано в главе 4.

Как правило, наборы применяют для управления группами элементов, динамически формируемых во время выполнения. Например, можно создать приложение для анализа группы объектов *Customer*, представляющих сведения о покупателях.

Создание набора объектов *Customer* позволит анализировать отдельные объекты, перебирать их в цикле, а также добавлять новые и удалять устаревшие объекты по мере необходимости.

Класс ArrayList

Класс *System.Collections.ArrayList* предоставляет базовую функциональность для работы с наборами, достаточную для решения многих задач. Этот класс позволяет динамически добавлять и удалять элементы из простого списка. Для извлечения элемента необходимо указать его индекс в списке. Новые экземпляры класса *ArrayList* создают так:

Visual Basic .NET

```
Dim myList As New System.Collections.ArrayList()
```

Visual C#

```
System.Collections.ArrayList myList = new System.Collections.ArrayList();
```

Метод Add

К созданному экземпляру класса *ArrayList* можно добавлять новые элементы при помощи метода *Add:*

Visual Basic .NET

```
Dim myWidget As New Widget()
myList.Add(myWidget)
```

Visual C#

```
Widget myWidget = new Widget();
myList.Add(myWidget);
```

Нумерация объектов-элементов набора *ArrayList* начинается с нуля, то есть первый элемент списка получает индекс «О», второй — «1» и т.д.

Свойство-индексатор Item

Свойство *Item* набора позволяет извлечь из него элемент с заданным индексом. В Visual Basic .NET свойство *Item* считается свойством набора по умолчанию. Таким образом, при обращении к элементам набора можно опускать имя свойства *Item,* как при обращении к элементу массива. В Visual C# свойства наборов по умолчанию называются *индексаторами* (indexer), поэтому вместо явного указания имени свойства *Item* в программах на C# необходимо применять ссылки на элементы массива. Подробнее свойства по умолчанию и индексаторы обсуждаются на следующем занятии. Следующий пример демонстрирует оба способа получения элемента набора посредством свойства *Item.*

Visual Basic .NET

```
Dim myObject As Object
myObject = myList(0)
' Следующая строка эквивалентна предыдущей.
myObject = myList.Item(0)
```

Visual C#

```
object myObject;
myObject = myList[0];
```

Все ссылочные типы, хранимые в наборах, имеют тип *Object.* Поэтому свойство *Item* всегда возвращает ссылку на *Object*, независимо от реального типа объекта, который находится в этом элементе. Если же нужно получить исходный тип, который был записан в набор, необходимо явно привести объект, возвращаемый свойством *Item,* к требуемому типу:

Visual Basic .NET

```
Dim myWidget As Widget
' Этот пример предполагает наличие набора объектов Widget.
myWidget = CType(widgetCollection(0), Widget)
```

Visual C#

```
Widget myWidget;
myWidget = (Widget)widgetCollection[0];
```

Методы Remove и RemoveAt

Эти методы предназначены для удаления элементов из набора. Метод *Remove* принимает в качестве параметра ссылку на объект, удаляемый из набора:

Visual Basic .NET

```
Dim myWidget As New Widget()
Dim myList As New System.Collections.ArrayList()
' Добавить в набора объект Widget.
myList.Add(myWidget)
' Удалить объект Widget из набора.
myList.Remove(myWidget)
```

Visual C#

```
Widget myWidget = new Widget();
System.Collections.ArrayList myList = new
    System. CollectionsArrayList();
// Добавить объект Widget в набор.
myList.Add(myWidget);
// Удалить объект Widget из набора.
myList.Remove(myWidget);
```

Попытка удалить из набора объект, отсутствующий в нем, не вызывает ошибку и просто игнорируется.

Метод *RemoveAt* позволяет удалить из набора элемент с заданным индексом, например, так:

Visual Basic .NET

```
Dim myList As New System.Collections.ArrayList()
' Добавить в набор три объекта Widget.
```

```
Dim X As Integer
For X = 1 to 3
   Dim aWidget As New Widget()
   myList.Add(aWidget)
Next
' Удалить объект Widget с индексом 1.
myList.RemoveAt(1)
```

Visual C#

```
System.Collections.ArrayList myList = new
   System.Collections.ArrayList();
// Добавить в набор три объекта Widget,
for (int x - 0; x < 3; x++)
{
   Widget myWidget - new Widget();
   myList.Add(myWidget);
}
// Удалить объект Widget с индексом 1.
myList.RemoveAt(1);
```

Заметьте: после удаления оставшиеся элементы набора нумеруются заново, что позволяет восстановить их последовательную нумерацию. В силу этих обстоятельств индекс элементов может изменяться.

Свойство Count

Свойство *Count* возвращает текущее число элементов набора. Поскольку нумерация элементов начинается с нуля, значение *Count* всегда на единицу больше верхней границы массива.

Другие типы наборов из класса Collections

В пространстве имен *System.Collections* есть и другие классы наборов, позволяющие упорядочивать объекты, объединяя их в группы. Эти классы предоставляют дополнительные возможности и позволяют решать специфические задачи, недоступные классу *ArrayList*. Отдельные классы кратко описаны в таблице 3-6.

Таблица 3-6. Члены класса *System.Collections*

Имя	Описание
BitArray	Представляет уплотненный массив однобитных элементов (принимающих значение 1 или 0)
CollectionBase	Основа для создания пользовательских классов наборов, предоставляет большинство функций, необходимых для обслуживания наборов
Hashtable	Набор пар «ключ — значение», упорядоченных по значению хеш-кода ключа
Queue	Набор объектов, организованный по принципу «первым вошел — первым вышел»
SortedList	Группа объектов, доступных через значение индекса или ключа
Stack	Набор объектов, организованный по принципу «первым вошел — последним вышел»

Перебор элементов массивов и наборов

В Visual Basic .NET и Visual C# имеется специальный синтаксис для перебора элементов массивов и наборов в цикле. Оператор *For Each (foreach)* позволяет поочередно обработать все элементы массива или набора. Это делается следующим образом:

Visual Basic .NET

```
Dim myArray() As Integer = {1,2,3,4,5}
Dim I As Integer
For Each I in myArray
   MessageBox.Show(I.ToString())
Next
```

Visual C#

```
int[] myArray - new int[] {1,2,3,4,5};
foreach (int I in myArray)
{
   MessageBox.Show(I.ToString());
}
```

При использовании такого синтаксиса с наборами у всех элементов набора должен быть тот же тип, что и у переменной цикла. Если в цикле обратиться к элементу набора, который невозможно записать в переменную цикла, генерируется исключение *InvalidCastException*. Во избежание подобных проблем при работе с наборами, содержащими объекты разных типов, переменную цикла следует объявлять с типом *Object*, а также проверять допустимость типа элемента набора с помощью оператора *typeof* (Visual Basic .NET) или метода *GetType* (Visual C#):

Visual Basic .NET

```
' Этот пример предполагает, что myList - это набор типа ArrayList,
' содержащий, помимо строк, объекты других типов.
Dim o as Object
For Each o in myList
   If TypeOf o Is String Then
      MessageBox.Show(o.ToString())
   End If
Next
```

Visual C#

```
// Этот пример предполагает, что myList - это набор типа ArrayList,
// содержащий, помимо строк, объекты других типов.
foreach (object o in myList)
{
   if (o.GetType() == typeof(string))
   {
      MessageBox.Show(o.ToString());
   }
}
```

Важно отметить, что при использовании цикла *For Each* (*foreach*),ссылка на элемент набора или массива, которая содержится в переменной цикла, доступна только для чтения. Следовательно, этот цикл не позволяет модифицировать элементы массивов и наборов. Чтобы получить возможность модифицировать элементы массива или набора при переборе их в цикле, следует использовать цикл *For...Next(for)* следующим образом:

Visual Basic .NET

```
Dim myArray() As Integer = {1,2,3,4,5}
Dim x As Integer
For x = 0 to myArray.GetUpperBound(0)
   myArray(x) += 1
   MessageBox.Show(myArray(x).ToString())
Next
```

Visual C#

```
int[] myArray = new int[] {1,2,3,4,5};
for (int x = 0; x <= myArray.GetUpperBound(0); x++)
{
   myArray[x] ++;
   MessageBox.Show(myArray[x].ToString());
}
```

Резюме

- Константы и перечисления делают код более читабельным и устойчивым к ошибкам, заменяя часто употребляемые значения понятными пользователю именами. Константы определяют с помощью ключевого слова *Const* (*const*);для объявления перечислений применяют ключевое слово *Enum (enum)*. Константы могут быть объявлены на основе любых типов данных, а перечисления — только на основе целочисленных типов.
- Массивы бывают одномерными и многомерными, последние в свою очередь могут быть прямоугольными и невыровненными. Все измерения прямоугольного массива составляют члены, образованные одинаковым числом элементов из других измерений. Члены измерений невыровненного массива отличаются числом образующих их элементов из других измерений, В обоих случаях чем больше у массива измерений, тем он сложнее.
- Наборы позволяют работать с группами объектов одного или разных типов, Классов наборов несколько, все они располагаются в пространстве имен *System, Collections*. Класс *System.Collections.ArrayList* предоставляет основную функциональность для работы с наборами, достаточную для решения типовых задач.
- Оператор *For Each (foreach)* позволяет перебирать значения элементов массива или коллекции в цикле, но не позволяет модифицировать значения элементов. Чтобы получить такую возможность, необходимо применить оператор *For...Next (for)*.

Занятие 3. Реализация свойств

Свойства — это члены классов, открывающие доступ к переменным или объектам класса. Свойства напоминают одновременно поля и методы. Синтаксис доступа к свойствам тот же, что и для полей, только при получении или установке свойства на самом деле вызывается специальный метод, который и выполняет запрошенное действие. У свойства также может быть код, проверяющий значение перед установкой свойства и выполняющий любые другие действия, необходимые приложению.

Изучив материал этого занятия, вы сможете:

- ✓ рассказать, как реализуют свойства;
- ✓ создавать свойства, доступные только для чтения или только для записи;
- ✓ объяснить, что такое свойство по умолчанию и индексатор и как их создать;
- ✓ сделать набор объектов доступным через свойство.

Продолжительность занятия - около 30 минут.

Реализация свойств

Свойства позволяют сделать доступ к значениям и объектам — членам класса, более устойчивым к ошибкам, чем при использовании обычных полей. В сущности, свойство — это специальный метод, внешне напоминающий поле. Значения свойств устанавливают и читают так же, как значения полей:

Visual Basic .NET

```
' Установить свойство TextBox1.Text
TextBox1.Text = "Text Property"
Dim myString As String
' Получить значение TextBoxl.Text и присвоить его переменной myString
myString = TextBoxl.Text
```

Visual C#

```
// Установить свойство TextBox1.Text
textBoxl.Text = "Text property";
string rnyString;
// Получить значение TextBoxl.Text и присвоить его переменной myString
myString = textBox1.Text;
```

На этом уровне свойства выглядят и работают, как поля, но код, который «стоит» за свойствами, несколько сложнее. Свойство состоит из двух специальных методов: *получателя* (getter) и *установщика* (setter) . Далее показан пример объявления свойства *MyText*:

Visual Basic .NET

```
' Создать закрытую локальную переменную для хранения значения свойства.
Private mText As String
```

```
' Реализация свойства
Public Property MyText() As String
   Get
      ' Вернуть значение локальной переменной.
      Return mText
   End Get
   Set(ByVal Value As String)
      ' Установить значение локальной переменной,
      mText = Value
   End Set
End Property
```

Visual C#

```
// Создать закрытую локальную переменную для хранения значения свойства.
private string mText;
// Реализация свойства
public string MyText
{
   get
   {
      // Вернуть значение локальной переменной.
      return mText;
   }
   set
   {
      // Установить значение локальной переменной.
      mText = value;
   }
}
```

В методе-установщике применяется специальное ключевое слово *Value* (*value*), представляющее новое значение, присваиваемое данному свойству. Свойства открывают доступ к объектам любого типа, а также к массивам и наборам. Объект или значение, предоставляемое свойством, обычно хранится в закрытой *(private)* локальной переменной (такой, как *m Text* из предыдущего примера); при получении свойства его значение извлекается из этой переменной.

Пример, показанный выше, демонстрирует довольно примитивное свойство, которое мало отличается от обычного поля. Одно из ключевых преимуществ свойств состоит в возможности размещения в методах, получающих и устанавливающих значение свойства, дополнительного кода, который выполняет необходимые вычисления или проверяет введенное значение. Ниже показан пример свойства, перед установкой которого новое значение проверяется.

Visual Basic .NET

```
Private mLength As Integer
Public Property Length() As Integer
   Get
      ' Вернуть значение локальной переменной.
```

```
      Return mLength
   End Get
   Set(ByVal Value As Integer)
      ' Проверяем, не превышает ли значение заданный предел (32).
      If Value > 32 Then
         MessageBox.Show(Value.ToString & _
            " is too large a value for this property!")
      Else
         mLength = Value
      End If
   End Set
End Property
```

Visual C#

```
private int mLength;
public int Length
{
   get
   {
      // Вернуть значение локальной переменной.
      returnmLength;
   }
   set
   {
      // Проверяем, не превышает ли значение заданный предел (32).
      if (value > 32)
         MessageBox.Show(value.ToString() +
            " is too large a value for this property!");
      else
         mLength = value;
   }
}
```

► **Как создать свойство**

1. Объявите закрытое поле или объект, в котором будет храниться значение (или объект), возвращаемое этим свойством.
2. Напишите для свойства код, возвращающий значение.
3. Добавить, если необходимо, дополнительный код проверки в метод, устанавливающий значение или получающий значение свойства.

Свойства, доступные только для чтения или только для записи

Иногда необходимо создать свойство, значение которого клиент получит, но не сможет модифицировать после того, как оно было установлено во время инициализации класса. Еще реже требуются свойства, которые не доступны для чтения, но доступны для модификации. Такие свойства называют соответственно свойствами, доступными *только для чтения* (read-only) и *только для записи* (write-only).

Создание свойств, доступных только для чтения

Процедура создания подобных свойств мало отличается от таковой для обычных свойств. Чтобы создать на Visual C# свойство, доступное только для чтения, достаточно определить свойство, у которого имеется метод *Get,* а метод *Set* отсутствует. Объявляя подобное свойство на Visual Basic .NET, необходимо пометить его ключевым словом *ReadOnly*, метод *Set* также следует опустить. В обоих языках закрытую переменную, в которой хранится значение свойства, необходимо объявлять с ключевым словом *ReadOnly (readonly).* Вот пример свойства, доступного только для чтения:

Visual Basic .NET

```
Private ReadOnly mInt as Integer
Public Readonly Property InstanceNumber() As Integer
   Get
      Return mInt
   End Get
End Property
```

Visual C#

```
private readonly int mInt;
public int InstanceNumber
{
   get
   {
      return mInt;
   }
}
```

Поскольку переменная, хранящая значение свойства, доступна только для чтения, ее нельзя модифицировать в коде. Начальное значение для такой переменной необходимо задавать в конструкторе класса.

▶ **Как создать свойство, доступное только для чтения**

1. Объявите закрытую переменную, в которой будет храниться значение свойства, пометив ее ключевым словом *ReadOnly (readonly).*
1. Напишите реализацию свойства, предоставив только метод для получения значения свойства, а устанавливающий его метод опустите. В Visual Basic .NET это свойство также необходимо пометить ключевым словом *ReadOnly.*
3. Добавьте к конструктору класса, членом которого является переменная свойства, код, присваивающий ей начальное значение.

Свойства, доступные только для записи

Можно создать свойство, значение которого клиент сумеет модифицировать, но не сможет прочитать, однако такие свойства требуются очень редко. Свойства, доступные только для записи, применяются для локализации форм. Например, если задать другой регион, код свойства соответствующим образом изменит форму, а читать эти свойства не обязательно.

Свойства, доступные только для записи, создаются практически так же, как свойства, доступные только для чтения. Для хранения значения этого свойства также требуется объявить закрытую локальную переменную, но в этом случае следует реализовать только метол, устанавливающий значение свойства, а метод, получающий его, опустить. В Visual Basic .NET подобное свойство необходимо пометить ключевым словом *WriteOnly*.

▶ **Как создать свойство, доступное только для записи**

1. Объявите закрытую переменную для хранения значения свойства.
2. Напишите реализацию свойства, предоставив только метод, устанавливающий значение свойства. В этот метод можно поместить любой код, который следует выполнить при изменении значения свойства. Кроме того, в Visual Basic .NET это свойство следует пометить ключевым словом *WriteOnly,*

Параметризованные свойства

Как правило, разработчики создают свойства, возвращающие единственное значение или объект. В Visual Basic .NET можно создавать свойства, которые принимают параметры. Как правило, такие свойства возвращают диапазон (или массив) значений. Рассмотрим для примера объект *Engine,* представляющий двигатель («engine» по-английски — двигатель). У реального двигателя может быть несколько цилиндров, аналогично объект *Engine* включает несколько объектов *Cylinder.* Чтобы не писать код для доступа к каждому из этих объектов, создают свойство *Cylinder*, получающее или устанавливающее значение любого объекта *Cylinder в* зависимости от переданного параметра.

Объявление параметризованных свойств не отличается от такового для обычного свойства, единственное отличие — в объявлении параметра. Далее пишут код методов, получающих и устанавливающих значение свойства. Вот пример параметризованного свойства, открывающего доступ к массиву объектов:

Visual Basic .NET

```
' Создать массив для хранения значений свойства.
Private mCylinders(7) As Cylinder
Public Property Cylinders(ByVal i As Integer) As Cylinder
   Get
      ' Проверить параметр, переданный при обращении к свойству.
      If I > 7 Then
         MessageBox.Show("Property value out of range!")
         Exit Property
   End If
      ' Если объект не существует, создать его.
      If mCylinders(i) Is Nothing Then
      mCylinders(i) - New Cylinder()
   End If
   Return mCylinders(i)
End Get
Set(ByVal Value As Cylinder)
    ' Проверить параметр, переданный при обращении к свойству.
   If i > 7 Then
```

```
        MessageBox.Show("Property value out of range!")
        Exit Property
    End If
    mCylinders(i) = Value
End Set
End Property
```

Visual C# не поддерживает параметризованные свойства, за исключением индексаторов, о которых пойдет речь в следующем разделе.

Свойства по умолчанию и индексаторы

Компоненту, написанному на Visual Basic .NET, допустимо назначить свойство по умолчанию. Оно непременно должно быть параметризованным. Обычно это свойство возвращает диапазон, массив или набор значений. Назначать объекту свойство по умолчанию имеет смысл, только если его основной функцией является хранение других объектов и предоставление к ним доступа.

Если некоторое свойство назначено свойством по умолчанию, можно опускать его имя и обращаться к нему просто по имени объекта, членом которого оно является. Так, следующие команды являются эквивалентными:

Visual Basic .NET

```
' Этот пример предполагает, что свойство Item
' является свойством по умолчанию для объекта myCollection.
myCollection.Item(2) = "Default properties!"
myCollection(2) = "Default properties!"
```

Поскольку для обращения к свойству по умолчанию достаточно указать имя объекта, у класса может быть не более одного свойства по умолчанию. В объявлении свойства по умолчанию указывают ключевое слово *Default*:

Visual Basic .NET

```
Public Default Property Item(I As Integer) As String
    ' Реализация опущена
End Property
```

В Visual C# эквивалентом свойств по умолчанию являются индексаторы. Индексатор — это особое свойство, позволяющее получать доступ к группе объектов, указывая только имя объекта, в котором реализован индексатор. В коде класса индексатор всегда фигурирует под именем *this*. Это свидетельствует о том, что для обращения к индексатору достаточно указать имя объекта, членом которого оно является. Вот пример простого индексатора:

Visual C#

```
// Создать массив для хранения значений,
// к которым предоставляет доступ индексатор
private int[] IntArray;
// Переменная индекса (в квадратных скобках) служит
// для указания нужного значения индексатора.
public int this[int index]
{
```

```
    get
    {
        return IntArray[index];
    }
    set
    {
        IntArray[index] = value;
    }
}
```

▶ **Как создать свойство по умолчанию в Visual Basic .NET**

Для этого следует создать параметризированное свойство, возвращающее объект, заданный переданным ему параметром. Это свойство следует объявлять с ключевым словом *Default.*

▶ **Как создать индексатор в Visual C#**

Чтобы создать индексатор, следует объявить свойство с **именем** *this,* назначить для него переменную индекса и написать код, возвращающий значение с заданным индексом.

Свойства-наборы

Если объект должен предоставлять доступ к набору однотипных элементов, число которых заранее неизвестно, можно сделать этот набор доступным через свойство объекта. Сделав набор объектов доступным через свойство класса, вы сможете управлять доступом к этим объектам и проверять значения, которые записываются в это свойство.

Реализуют свойство-набор по-разному. Выбор способа зависит от необходимой функциональности. Простое свойство-набор возвращает набор объектов, при этом клиент может добавлять и удалять элементы из набора, а также вызывать любые методы набора. Чтобы запретить клиенту запись в свойство-набор новых экземпляров этого класса набора, сделайте это свойство доступным только для чтения - это не препятствует модификации объектов набора посредством его методов, Вот пример простого свойства-набора:

Visual Basic .NET

```
Private Readonly mWidgets As New System.Collections.ArrayList()
Public Readonly Property Widgets() As System.Collections.ArrayList
    Get
        Return mWidgets
    End Get
End Property
```

Visual C#

```
private readonly System.Collections.ArrayList mWidgets = new
    System.Collections.ArrayList();
public System.Collections.ArrayList Widgets
{
    get
```

```
    {
        return mWidgets;
    }
}
```

Хотя этот способ прост и позволяет получать доступ к элементам набора, у него есть один существенный недостаток. Большинство наборов, представленных классами из пространства имен *System. Collections,* принимает и возвращает свои элементы в виде объектов типа *Object.* Это означает, что можно добавить в набор (или удалить из него) объект любого типа. Иными словами, при добавлении к набору новых элементов проверка типов не выполняется, а возвращаемые элементы набора необходимо явно приводить к нужному типу. Впрочем, если свойство должно работать с разнородными объектами, этот подход приемлем.

Для создания простого свойства-набора достаточно реализовать доступное только для чтения свойство, возвращающее нужный набор.

Чтобы получить свойство-набор, более устойчивое к ошибкам, следует сделать набор доступным через свойство и написать код, выполняющий явное преобразование типов и проверку значений, присваиваемых свойству. Это позволит получать из набора элементы с заданным индексом и проверять типы помещаемых в него объектов. Такое свойство в Visual Basic .NET реализуют в виде параметризованного свойства, например, так:

Visual Basic .NET

```
Private mWidgets as New System.Collections.ArrayList()
Public Property Widget(I As Integer) As Widget
    Get
        Return CType(mWidgets(I), Widget)
    End Get
    Set(ByVal Value As Widget)
        mWidgets(I) = Value
    End Set
End Property
```

Visual C# не поддерживает параметризованные свойства, поэтому при использовании этого языка такие свойства реализуют как индексаторы либо в виде пары специальных методов (см. пример кода далее).

Visual C#

```
private System.Collections.ArrayList mWidgets = new
    System.Collections.ArrayList();
public Widget GetWidget(int I)
{
    return (Widget)mWidgets[I];
}
public void SetWidget(int I, Widget Wid)
{
    mWidgets[I] = Wid;
}
```

Аналогичным образом создают строго типизированные свойства, открывающие доступ к набору однотипных элементов, однако этот способ не лишен недостатков. Например, при его использовании нельзя применять оператор *For Each...Next (foreach)* для перебора элементов набора в цикле, а для всех методов данного свойства, которые должны быть доступными клиенту, потребуется написать оболочку. Наконец, в этом случае невозможны некоторые отступления от правил синтаксиса, разрешенные в Visual C#.

▶ **Как создать свойство-набор с проверкой типов**

- в Visual Basic .NET для этого следует создать параметризированное свойство, возвращающее элемент закрытого *(private)* набора, заданный индексом, и написать код, преобразующий возвращаемый объект в требуемый тип при помощи функции *CType*;
- в Visual C# для этого необходимо создать пару методов, получающих и устанавливающих элемент закрытого набора, заданный индексом, причем метод *get* должен явно приводить возвращаемый объект к требуемому типу.

Чтобы создать свойство с еще более строгой реализацией, можно объявить собственный класс строго типизированного набора на основе класса *System.Collections.CollectionBase*. Этот класс предоставляет основную функциональность, необходимую для обслуживания наборов, позволяя создавать собственную реализацию для свойства *item*. Созданный таким образом строго типизированный набор удается сделать доступным непосредственно через свойство (этот прием проиллюстрирован в приведенном выше примере); при этом вся функциональность наборов останется доступной. Помимо приема только элементов заданного типа, такой набор поддерживает любую другую необходимую функциональность. Наследование рассматривается в деталях в главе 4, там же вам предлагается упражнение с примером, иллюстрирующим создание строго типизированного набора.

Резюме

- Свойства открывают доступ к членам класса (переменным и объектам), позволяют создавать код, проверяющий значение свойства при обращении к нему и выполняющий ряд других действий. Свойства, доступные только для чтения или только для записи, позволяют ограничить доступ клиента к свойствам чтением или записью. Visual Basic .NET также позволяет создавать свойства, принимающие параметры и возвращающие диапазоны значений.
- К значению свойства по умолчанию разрешено обращаться, не указывая имя этого свойства. Такие свойства должны быть параметризированными. В Visual C# эквивалентом свойств по умолчанию являются индексаторы.
- Свойства-наборы открывают доступ к группам объектов, поддерживаемых классом. Предусмотрено несколько способов реализации подобных свойств, каждый из них обладает своими достоинствами и недостатками:
  - простейшее свойство-набор, возвращающее объект-набор из закрытой переменной; этот способ наименее устойчив к ошибкам;
  - свойство, возвращающее элемент закрытого набора с заданным индексом. Этот вариант более устойчив к ошибкам, но менее гибок;
  - класс строго типизированного набора. Применение этого способа требует от разработчика наибольших усилий, обеспечивая взамен доступ к строго типизированным наборам. Этот способ максимально устойчив к ошибкам.

Занятие 4. Реализация событий и делегатов

События уведомляют о важных происшествиях, случившихся в других частях приложения. Если где-то в приложении генерируется событие, остальные компоненты приложения получают возможность отреагировать на него исполнением методов, которые называются *обработчиками события*. В главе 2 мы показали, как писать в Visual Studio .NET обработчики для событий, генерируемых формами и элементами управления. На этом занятии вы научитесь объявлять и генерировать в приложении собственные события, писать для них обработчики, а также динамически подключать и отключать последние во время выполнения.

Изучив материал этого занятия, вы сможете:

- ✓ объяснить, что такое делегаты и как их создают;
- ✓ объявлять и генерировать события в приложении;
- ✓ создавать обработчики событий и связывать их с событиями;
- ✓ динамически подключать и отключать обработчики событий во время выполнения.

Продолжительность занятия - около 30 минут.

События являются членами класса, которые позволяют уведомлять другие компоненты приложения о том, что случилось с этим классом. Когда в приложении происходит что-то важное, оно *генерирует* событие, уведомляющее о случившемся все заинтересованные компоненты. Вместе с событием зачастую передаются различные аргументы, содержащие дополнительные сведения о нем. Другие части приложения *обрабатывают* это событие, то есть исполняют связанный с ним метод. Сигнатура метода, обрабатывающего событие, должна совпадать с сигнатурой этого события. Иными словами, метод должен принимать аргументы, которые передаются с событием. Одно событие может обрабатываться несколькими методами, а один метод может обрабатывать несколько событий.

Делегаты

Делегаты — это особые классы, обеспечивающие работу событий. В сущности, делегат — это указатель на функцию, обеспечивающий контроль типов. Он позволяет передавать ссылку на точку входа метода и исполнять этот метод без традиционного явного вызова. При объявлении делегата указывают сигнатуру вызываемого им метода и тип возвращаемого результата:

Visual Basic .NET

```
Public Delegate Function myDelegate(ByVal D As Double) As Integer
```

Visual C#

```
public delegate int myDelegate(double D);
```

В этом примере объявляется делегат *myDelegate*, вызывающий методы, которые возвращают результат типа *Integer (int)* и принимают в качестве параметра значение типа *Double (double)*. Чтобы воспользоваться делегатом, необходимо создать его экземпляр с указанием метода, который он будет вызывать. В Visual C# этот метод

задают по имени, а в Visual Basic .NET применяют оператор *AddressOf*, который создает ссылку на адрес метода. Вот соответствующие примеры:

Visual Basic .NET

```
' Это метод, который будет вызывать делегат.
Public Function ReturnInt(ByVal d As Double) As Integer
' Реализация метода опущена,
End Function
' Создать экземпляр делегата myDelegate,
' который будет вызывать метод ReturnInt.
Dim aDelegate As New myDelegate(addressof ReturnInt)
```

Visual C#

```
// Это метод, который будет вызывать делегат.
public int ReturnInt(double D)
{
   // Реализация метода опущена
}
// Создать экземпляр делегата myDelegate,
// который будет вызывать метод ReturnInt.
public void amethod()
{
    myDelegate aDelegate = new myDelegate(ReturnInt);
}
```

При создании делегата для метода экземпляра в Visual C# необходимо инициализировать делегат внутри этого метода. Делегаты статических методов можно инициализировать вне метода, который они вызывают.

После объявления и инициализации делегат можно вызывать, как обычную функцию, чтобы исполнить связанный с ним метод, например:

Visual Basic .NET

```
aDelegate(12345)
```

Visual C#

```
aDelegate(12345);
```

▶ **Как создать и вызвать делегат**

1. Объявите делегат с сигнатурой, идентичной сигнатуре метода (или методов), для вызова которого он будет использоваться.
2. Создайте экземпляр делегата, ссылающийся на метод с подходящей сигнатурой.
3. Вызывайте делегат в коде, ссылаясь на него по имени.

Объявление и генерация событий

В главе 2 говорилось, что у форм и элементов управления есть члены-события, которые генерируются в соответствующих обстоятельствах. Например, событие *Click*

генерируется в ответ на щелчок мышью. В классах можно объявлять собственные события и генерировать их при необходимости.

Объявление событий имеет непосредственное отношение к делегатам. В Visual C# необходимо явно указать тип-делегат, который будет использовать объявляемое событие. В Visual Basic .NET этот механизм по большей части работает «за кулисами»; при объявлении события достаточно указать его имя и сигнатуру обрабатывающего его метода — в результате этих действий создается делегат по умолчанию с заданной сигнатурой.

▶ **Как объявить событие на Visual Basic .NET**

Чтобы объявить событие, необходимо указать ключевое слово *Event* и сигнатуру методов, обрабатывающих это событие. При этом можно использовать любые модификаторы доступа: *Public*, *Private* или *Protected*. Вот пример объявления события:

Visual Basic .NET

```
Public Event CalculationComplete(ByVal Total As Double)
```

▶ **Как объявить событие на Visual C#**

Чтобы объявить событие, необходимо указать ключевое слово *Event* и тип-делегат, который будет использоваться этим событием. При этом разрешено применять любые модификаторы доступа: *public*, *private* или *protected*. Вот пример объявления события:

Visual C#

```
public delegate void calculationDelegate(double d);
public event calculationDelegate CalculationComplete;
```

Объявив событие, допустимо генерировать его в коде в ответ на соответствующие обстоятельства. Например, компонент, представляющий банковский счет, будет генерировать событие *Overdrawn* всякий раз, когда баланс счета опустится ниже нуля.

▶ **Как генерировать событие в Visual Basic .NET**

Для генерации события следует указать ключевое слово *RaiseEvent*, за которым следует имя генерируемого события и все необходимые параметры (в скобках), например:

Visual Basic .NET

```
RaiseEvent CalculationComplete(66532)
```

▶ **Как генерировать событие в Visual C#**

Чтобы генерировать событие, достаточно указать его имя (как при вызове метода) и все необходимые параметры (в скобках), например:

Visual C#

```
CalculationComplete(66532);
```

Реализация обработчиков событий

Чтобы генерировать объявленное событие, необходимо связать с ним один или несколько обработчиков. Обработчик события — это метод, который вызывается посредством делегата всякий раз, когда генерируется соответствующее событие. Чтобы этот механизм работал, необходимо связать обработчики с обрабатываемыми событиями. Если в программе на Visual Basic .NET генерировать событие, с которым не связан ни один обработчик, попросту ничего не произойдет; в Visual C# же подобные действия закончатся ошибкой.

Подобно другим членам, события бывают как членами экземпляров, так и общими *(Shared)* или статическими (*static*) членами. При создании обработчика для события экземпляра нужно связать метод-обработчик и это событие, для чего необходимо сначала создать объект. Общие (статические) события принадлежат классу, а не его экземплярам, поэтому для создания обработчика такого события не требуется предварительно создавать экземпляр класса.

Создание обработчиков событий в Visual Basic .NET

Обработчик события в Visual Basic .NET следует объявлять как процедуру (*Sub*), он не может возвращать значения. Процедуру связывают с обрабатываемым ей событием при помощи ключевого слова *AddHandler*. После *AddHandler* следует имя события и указатель на метод, с которым будет связано это событие. Указатель на метод создают оператором *AddressOf,* например, так:

Visual Basic .NET

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As _
   System.EventArgs)
   AddHandler myEvent, AddressOf myHandler
End Sub
```

Ключевое слово *AddHandler не* разрешается применять декларативно — только внутри метода. В силу этих причин связь между событием и его обработчиком создается только во время выполнения. Чтобы связь между обработчиком и событием существовала на протяжении всего срока жизни объекта, следует поместить ключевое слово *AddHandler в* конструктор объекта.

В Visual Basic .NET можно связывать обработчики с обрабатываемыми ими событиями и во время разработки. Для этого применяется конструкция *Handles*. При этом объект-приемник события необходимо объявить с ключевым словом *WithEvents*, которое разрешает ему регистрироваться для уведомления о событиях объекта, указанного в поле *WithEvents*.

Конструкция *Handles* завершает объявление метода, в ней определяют событие объекта, обрабатываемое этим методом, например:

Visual Basic .NET

```
Public Sub DisplayResults() Handles Account.CalculationComplete
' Реализация опущена
End Sub
```

Создание обработчиков событий на Visual C#

В отличие от Visual Basic .NET, в Visual C# обработчикам событий разрешается возвращать значение, которое можно присваивать переменной, как это делается при вызове функции. Чтобы связать некоторый метод с событием, следует создать экземпляр соответствующего делегата, ссылающегося на этот метод, и связать его с событием при помощи оператора +=. Этот оператор также позволяет связать событие с существующим экземпляром делегата, например, так:

Visual C#

```
// Это пример предполагает наличие метода DisplayResults
// с сигнатурой делегата calculationDelegate.
// Создать экземпляр делегата, он необходим, чтобы
// связать событие с его обработчиком.
Account.CalculationComplete += new calculationDelegate(DisplayResults);
// Связать событие с обработчиком при помощи существующего экземпляра делегата.
CalculationDelegate calc = new calculationDelegate(DisplayResults);
Account.CalculationComplete += calc;
```

Для событий элементов управления и классов из библиотеки базовых классов .NET Framework предоставляются делегаты по умолчанию. Чтобы добавить обработчик для этих событий вручную, необходимо объявить новый делегат либо создать новый экземпляр предопределенного типа-делегата. Например, с большинством элементов управления из пространства имен *System, WindowsForms* применяется класс-делегат *System, EventHandler*. Ниже показан пример, иллюстрирующий назначение обработчика событию *Click* элемента управления button 1:

Visual C#

```
button1.Click += new System.EventHandler(clickHandler);
```

В отличие от Visual Basic .NET, в Visual C# нет механизма для декларативного связывания событий с их обработчиками.

▶ **Обработка событий в Visual Basic .NET**

- При помощи ключевого слова *AddHandler* свяжите событие с методом, который его будет обрабатывать; для получения указателя на метод воспользуйтесь оператором *AddressOf*. Помните, что сигнатуры события и его обработчика должны совпадать.
- Другой вариант — связать метод с обрабатываемым им событием во время разработки (при помощи ключевого слова *Handles)*. При этом метод-обработчик должен быть членом объекта, объявленного с ключевым словом *WithEvents.*

▶ **Обработка событий в Visual C#**

Создайте экземпляр делегата, соответствующего событию, и ссылающегося на метод, который должен обрабатывать это событие, затем воспользуйтесь оператором +=, чтобы связать событие с созданным экземпляром делегата.

Создание обработчика для группы событий

Можно создать метод, способный обрабатывать несколько событий. Обычно это требуется, если несколько экземпляров класса или элементов управления генерируют одинаковые события. Например, если на форме расположена группа кнопок,

выполняющих сходные функции в приложении, стоит создать единственный метод, который будет обрабатывать событие *Click* для всех этих кнопок. Кнопку, генерировавшую событие, определяют по значению параметра *sender*.

Связать обработчик с несколькими событиями просто: это делается при помощи оператора *AddHandler (в* Visual Basic .NET, а в C# — +=), как и в случае с одним событием.

Visual Basic .NET

```
' Этот пример предполагает наличие метода ClickHandler с
' сигнатурой, подходящей для приема событий Click.
AddHandler Button1.Click, AddressOf ClickHandler
AddHandler Button2.Click, AddressOf ClickHandler
```

Visual C#

```
// Этот пример предполагает наличие метода ClickHandler с
// сигнатурой, подходящей для приема событий Click.
button1.Click += new System.EventHandler(ClickHandler);
button2.Click += new System.EventHandler(ClickHandler);
```

В Visual Basic .NET для связывания одного обработчика с несколькими событиям применяют конструкцию *Handles,* в которой перечисляют (через запятую) все события, с которыми следует связать обработчик:

Visual Basic .NET

```
Private Sub ClickHandler (ByVal sender As System.Object, e As _
   System.EventArgs) Handles Button1.Click, Button2.Click, _
   Button3.Click
   ' Реализация опущена.
End Sub
```

События с несколькими обработчиками

У события может быть несколько обработчиков. При генерации такого события, все обработчики исполняются в порядке их связывания с этим событием. Так, если с событием x были поочередно связаны в коде обработчики y, z и q, то при генерации этого события сначала исполняется обработчик y, затем z и в последнюю очередь q. В Visual C# имеется одна особенность: если событие возвращает значение, при наличии у него нескольких обработчиков, будет возвращен результат исполнения последнего обработчика.

Чтобы связать с событием несколько обработчиков, достаточно поочередно подключить их к нужному событию. В Visual Basic .NET это делается декларативно с помощью конструкции *Handles.* Правда, в этом случае трудно предсказать порядок вызова обработчиков при генерации события; если важно его знать, лучше определить его экспериментально.

Отключение обработчиков во время выполнения

Как уже говорилось, для динамического связывания обработчика с событием применяется ключевое слово *AddHandler* (в Visual Basic .NET) или оператор += (в Visual C#). Возможна и обратная операция — динамический разрыв связи между обработ-

чиком и событием во время выполнения. Рассмотрим следующий пример. Допустим, имеется класс, представляющий банковский счет, который генерирует событие *NotSufficientFunds* всякий раз, когда владелец счета пытается выписать чек на сумму, превышающую остаток средств на счете. С этим событием можно связать два обработчика: метол *Charge Account,* вычисляющий размер сбора за предосгавление дополнительного кредита в размере суммы, на которую выписан чек, и *Notify,* извещающий владельца счета о превышении кредита (об овердрафте). Метод *ChargeAccount* имеет смысл вызывать каждый раз, когда проходит овердрафтный чек, а метол *Notify* достаточно вызвать однажды, а когда баланс счета вновь станет положительным, можно его вызвать еще раз. Чтобы реализовать такой сценарий, следует разорвать связь между методом *Notify* и событием *NotSufficientFunds* после первого вызова, и восстановить ее, когда баланс счета станет положительным.

Методика удаления связи между обработчиком и событием похожа на методику создания такой связи. В Visual Basic. NET для отключения обработчика события от делегата применяют ключевое слово *RemoveHandler,* указывая адрес обработчика с помощью оператора *AddressOf.* В Visual C# для динамического отключения события от его обработчика служит оператор -=. Этот оператор требует указать ссылку на экземпляр делегата с соответствующей сигнатурой и метод, который необходимо отключить от события. Ниже показан пример отключения метода *Display Results* от события *Account. Calculation Complete:*

Visual Basic .NET

```
RemoveHandler Account.CalculationComplete, AddressOf DisplayResults
```

Visual C#

```
Account.CalculationComplete -= new calculationDelegate(DisplayResults);
```

Резюме

- События — это члены классов, позволяющие им обмениваться сведениями о важных событиях. Экземпляр класса может генерировать событие — член этого класса, чтобы уведомить другие компоненты приложения. Событие обрабатывается методами, зарегистрированными как обработчики этого события. Обработчики выполняются каждый раз, когда генерируется событие.
- Работу событий обеспечивают делегаты. При объявлении события в Visual Basic .NET достаточно указать само событие, а в Visual C# дополнительно требуется указать подходящий делегат, В Visual Basic .NET события генерируют при помощи оператора *RaiseEvent, а* в Visual C# для этого необходимо обратиться к событию по имени. В Visual Basic .NET обработчики событий не способны возвращать значение, а в Visual C# — могут.
- Метод, обрабатывающий событие, называется его обработчиком. Чтобы программно установить связь между обработчиком и событием, которое он обрабатывает, воспользуйтесь экземпляром соответствующего делегата. В Visual Basic .NET для этого используют ключевое слово *AddHandler* (а для получения ссылки на метод — оператор *AddressOf).* В Visual C# события связывают с делегатами при помощи оператора +=. В Visual Basic .NET такую связь можно создать декларативно посредством конструкции *Handles.*
- У события бывает несколько обработчиков; и наоборот, один обработчик способен обрабатывать несколько событий. Обработчики событий можно динамически подключать и отключать во время выполнения.

Практикум 3. Добавление компонентов и реализация членов

Вы создадите класс *Doughnut Machine,* представляющий аппарат по изготовлению пончиков для виртуальной фабрики, речь о которой шла в практикуме предыдущей главы. Новый класс должен по графику создавать экземпляры класса *Doughnut,* обновляя форму интерфейса при помощи событий. Решение задач этого практикума хранится в каталоге \Labs\Ch03\Solution на прилагаемом к книге компакт-диске.

Прежде всего

Для выполнения задач практикума вам понадобятся результаты практикума главы 2; можно также воспользоваться готовым решением, загрузив его с прилагаемого компакт-диска.

Продолжительность практикума - около 45 минут.

Упражнение 1. Создание компонента DoughnutMachine

Сейчас вы создадите компонент, представляющий аппарат для производства пончиков. Этот класс создает экземпляры класса *Doughnut* и управляет ими, а также открывает доступ к набору объектов, представляющих разные виды пончиков. У него также будут события, уведомляющие компонент-контейнер о том, что пончик готов.

▶ **Создание компонента DoughnutMachine**

1. В меню Project выберите команду Add Component — откроется окно Add New Item. Назовите новый компонент **DoughnutMachine.vb** или **DoughnutMachine.cs** в соответствии с используемым языком программирования и щелкните OK — компонент будет открыт в окне дизайнера.
2. На панели Toolbox перейдите на вкладку Windows Forms и дважды щелкните компонент Timer — этот компонент *Timer* появится в окне дизайнера.
3. В окне Solution Explorer щелкните правой кнопкой объект *DoughnutMachine* и выберите View Code.
4. Добавьте к компоненту *DoughnutMachine* перечисление, представляющее различные сорта пончиков. Оно выглядит примерно так:

Visual Basic .NET

```
Public Enum DoughnutType
   Glazed
   Sugar
   Chocolate
   ChocolateCake
   Custard
   Grape
   Lemon
```

```
    PlainCake
    SugarCake
End Enum
```

Visual C#

```
public enum DoughnutType
{
    Glazed,
    Sugar,
    Chocolate,
    ChocolateCake,
    Custard,
    Grape,
    Lemon,
    PlainCake,
    SugarCake
}
```

5. Добавьте к созданному классу свойство, представляющее сорт пончика, который он «печет» в данный момент. Для этого объявите в нем закрытую переменную для хранения значения этого свойства. Новое свойство класса будет возвращать значение перечислимого типа *DoughnutType*. Вот код этого свойства:

Visual Basic .NET

```
Private mFlavor As DoughnutType
Public Property Flavor() As DoughnutType
    Get
        Return mFlavor
    End Get
    Set (ByVal Value As DoughnutType)
        mFlavor = Value
    End Set
End Property
```

Visual C#

```
private DoughnutType mFlavor;
public DoughnutType Flavor
{
    get
    {
        return mFlavor;
    }
    set
    {
        mFlavor = value;
    }
}
```

6. **Объявите внутри** класса ***DoughnutMachine*** класс ***Doughnut,*** **который** будет представлять **пончик. У него** должны **быть** свойства *flavor* **(сорт пончика),** ***price*** (его цена), а также свойство, **доступное только для чтения, — для хранения** времени **изготовления пончика. Поскольку единственное место,** где **разрешается>становить свойство,** доступное **только для чтения, — это** конструктор, **необходимо** написать для **класса** ***Doughnut*** **конструктор,** записывающий **в это свойство текущее время на момент создания** экземпляра **пончика. Код класса может** быть таким:

Visual Basic .NET

```
Public Class Doughnut
    ' Это переменные для хранения значений свойств.
   Private mFlavor As DoughnutType
    ' Цена по умолчанию,
   Private mPrice As Single - .50
   Private Readonly mTirneOfCreation As Date
    ' Это свойства класса,
   Public Property Flavor() As DoughnutType
      Get
         Returr mFlavor
      End Get
      Set(ByVal Value As DoughnutType)
         mFlavor - Value
         End Set
      End Property
      Public Property Price() As Single
         Get
         Return mPrice
      End Get
      Set(ByVal Value As Single)
         mPrice = Value
         End Set
      End Property
      Public Readonly Property TimeOfCreation() As Date
      Get
         Return mTimeOfCreation
      End Get
      End Property
      ' Это конструктор, в котором устанавливается значение
      ' свойства mTimeOfCreation.
      Public Sub New(ByVal Flavor As DoughnutType)
         ' Date.Now - это свойство класса Date,
         ' возвращающее текущее время.
         mTimeOfCreation = Date. Now
         mFlavor = Flavor
      End Sub
   End Class
```

Visual C#

```
public class Doughnut
{
   // Это переменные для хранения значений свойств.
   private DoughnutType mFlavor;
   // Цена по умолчанию.
   private float mPrice - .50F;
   private readonly System.DateTime mTimeOfCreation;
   // Это свойства класса,
   public DoughnutType Flavor
   {
      get
      {
         return mFlavor;
      }
      set
      {
         mFlavor = value;
      }
   }
   public float Price
   {
      get
      {
         return Price;
      }
      set
      {
         mPrice - value;
      }
   }
   public System.DateTime TimeOfCreation
   {
      get
      {
         return mTimeOfCreation;
      }
   }
   // Это конструктор, в котором устанавливается значение
   public Doughnut (DoughnutType Flavor)
   {
      // System.DateTime.Now - это свойство класса System.DateTime,
      // возвращающее текущее время
      mTimeOfCreation = System.DateTime.Now;
      mFlavor = Flavor;
   }
}
```

7. Объявите в классе *DoughnutMachine* закрытый набор и свойство по умолчанию (в Visual Basic .NET, в Visual C# его нужно заменить индексатором), открывающее доступ к элементам набора. Созданный набор хранит экземпляры класса *Doughnut,* созданные объектом *Doughnut Machine,* Вот код этого свойства:

Visual Basic .NET

```
Private mDoughnuts As New System.Collections.ArrayList()
Public Default Property Doughnuts(Index As Integer) As Doughnut
    Get
        Return CType(mDoughnuts(Index), Doughnut)
    End Get
    Set (Value As Doughnut)
        mDoughnuts(Index) = Value
    End Set
End Property
```

Visual C#

```
private System.Collections.ArrayList rnDoughnuts = new
    System.Collections.ArrayList();
public Doughnut this[int Index]
{
    get
    {
        return (Doughnut)mDoughnuts[Index];
    }
    set
    {
        mDoughnuts[Index] = value;
    }
}
```

8. Добавьте к классу открытое (*public*) событие. Оно сгенерируется к моменту готовности очередного пончика (то есть когда будет создан представляющий его объект). В Visual C# следует также объявить открытый делегат,

Visual Basic .NET

```
Public Event DoughnutComplete()
```

Visual C#

```
public delegate void DoughnutCompleteDelegate();
public event DoughnutCompleteDelegate DoughnutComplete;
```

9. При помощи раскрывающегося списка в окне редактора кода (Visual Basic NET) или кнопки Events в окне дизайнера (Visual C#) добавьте обработчик события *Timer1.Tick* (см. код ниже), который создает новые объекты *Doughnut,* добавляет их к набору *mDoughnuts* и генерирует событие *DoughnutComplete.*

Visual Basic .NET

```
Private Sub Timer1_Tick(ByVal sender As Object, ByVal e As _
   System, EventArgs) Handles Timer1.Tick
   Din aDoughnut As New Doughnut(Me.Flavor)
   mDoughnuts.Add(aDoughnut)
   RaiseEvent DoughnutComplete()
End Sub
```

Visual C#

```
private void timer1_Tick(object sender, System.EventArgs e)
{
   Doughnut aDoughnut = new Doughnut(this.Flavor);
   mDoughnuts.Add(aDoughnut);
   DoughnutComplete();
}
```

10. Создайте два свойства, доступных только для записи: *Enabled* и *Interval* с типами *Boolean* и *Integer* соответственно — они потребуются для установки внутренних свойств таймера. Вот код этих свойств:

Visual Basic .NET

```
Public WriteOnly Property EnabledC) As Boolean
   Set(ByVal Value As Boolean)
      Timer1.Enabled = Value
   End Set
End Property
Public WriteQnly Property Interval() As Integer
   Set(ByVal Value As Integer)
      Timer1.Interval - Value
   End Set
End Property
```

Visual C#

```
public bool Enabled
{
   set
   {
      timer1.Enabled - value;
   }
}
public int Interval
{
   set
   {
      timer1.Interval = value;
   }
}
```

11. Добавьте к классу *DoughnutMachine* метод, настраивающий виртуальный аппарат, чтобы тот «пек» нужный сорт пончиков через заданные интервалы, и включающий этот аппарат. Этот метод должен принимать в качестве параметра объект, представляющий сорт пончика (*DoughnutType*). Вот этот метод:

Visual Basic .NET

```
Public Sub MakeDoughnuts(ByVal dFlavor As DoughnutType)
   Flavor = dFlavor
   Select Case dFlavor
      Case DoughnutType.Chocolate
         Interval = 15000
      Case DoughnutType.ChocolateCake
         Interval = 12000
      Case DoughnutType.Custard
         Interval = 10000
      Case DoughnutType.Glazed
         Interval = 10000
      Case DoughnutType.Grape
         Interval = 10000
      Case DoughnutType.Lemon
         Interval = 10000
      Case DoughnutType.PlainCake
         Interval = 5000
      Case DoughnutType.Sugar
         Interval = 8000
      Case DoughnutType.SugarCake
         Interval = 6000
   End Select
   Enabled - True
End Sub
```

Visual C#

```
public void MakeDoughnuts(DoughnutType dFlavor)
{
   Flavor = dFlavor;
   switch(dFlavor)
   {
      case DoughnutType.Chocolate:
         Interval = 15000;
         break;
      case DoughnutType.ChocolateCake:
         Interval = 12000;
         break;
      case DoughnutType.Custard:
         Interval = 10000;
         break;
      case DoughnutType.Glazed:
```

```
            Interval = 10000;
            break;
         case DoughnutType.Grape:
            Interval = 10000;
            break;
         case DoughnutType.Lemon:
            Interval = 5000;
            break;
         case DoughnutType.PlainCake:
            Interval = 5000;
            break;
         case DoughnutType.Sugar:
            Interval = 8000;
            break;
         case DoughnutType.SugarCake:
            Interval = 6000;
            break;
      }
      Enabled = true;
   }
```

Упражнение 2. Подключение класса DoughnutMachine к пользовательскому интерфейсу

Сейчас вы добавите к *классу frmMain* код для создания экземпляра класса *DoughnutMachine,* выпекающего пончики для виртуального магазина, а также код для выбора сорта пончиков и метод, обрабатывающий событие *DoughnutComplete*, генерируемое компонентом *DoughnutMachine.* Наконец, вы добавите метод, останавливающий производство пончиков.

1. Объявите на уровне класса переменную, представляющую экземпляр компонента *DoughnutMachine:*

Visual Basic .NET

```
Private myDoughnutMachine As DoughnutMachine
```

Visual C#

```
private DoughnutMachine myDoughnutMachine;
```

2. Напишите обработчик события *frmMain_Load*, который создает экземпляр класса *DoughnutMachine* и помещает его в переменную *myDoughnutMachine.* Вот примерный код такого обработчика

Visual Basic .NET

```
Private Sub frmMain_Load(ByVal sender As System.Object, ByVal e As _
   System.EventArgs) Handles MyBase.Load
   myDoughnutMachine = New DoughnutMachine()
End Sub
```

Visual C#

```
private void frmMain_Load(object sender. System.EventArgs e)
{
   myDoughnutMachine = new DoughnutMachine();
}
```

3. Объявите закрытые переменные для учета количества пончиков:

Visual Basic .NET

```
Private mRaisedGlazed As Integer
Private mRaisedSugar As Integer
Private mRaisedChocolate As Integer
Private mCakePlain As Integer
Private mCakeChocolate As Integer
Private mCakeSugar As Integer
Private mFilledLemon As Integer
Private mFilledGrape As Integer
Private mFilledCustard As Integer
```

Visual C#

```
private int mRaisedGlazed;
private int mRaisedSugar;
private int mRaisedChocolate;
private int mCakePlain;
private int mCakeChocolate;
private int mCakeSugar;
private int mFilledLemon;
private int mFilledGrape;
private int mFilledCustard;
```

4. Создайте для элемента Make из меню Doughnuts вложенные элементы Raised, Cake и Filled, присвоив им внутренние имена *mnuRaised*, *mnuCake* и *mnuFilled* соответственно. Для элемента Raised создайте вложенные элементы Glazed, Sugar и Chocolate, задав для них имена *mnuRaisedGlazed, mnuRaisedSugar* и *mnuRaisedChocolate*. Вид готового меню в окне дизайнера показан на рис. 3-1.
5. Добавьте к обработчику события *Click* элемента меню *mnuRaisedGlazed* код, вызывающий метод *DoughtnutMachine.MakeDoughnut* с параметром, определяющим сорт пончика. Этот код также должен устанавливать свойство *Checked* элемента меню *mnuRaisedGlazed* в *true,* а аналогичные свойства пунктов *mnuRaisedSugar* и mnuRaisedChocolate — в *false*. Вот этот код:

Visual Basic .NET

```
Private Sub mnuRaisedGlazed_Click(ByVal sender As _
   System.Object, ByVal e _
   As System.EventArgs) Handles mnuRaisedGlazed.Click
   mnuRaisedGlazed.Checked = True
   mnuRaisedSugar.Checked - False
   mnuRaisedChocolate.Checked = False
```

```
    myDoughnutMachine.MakeDoughnuts(DoughnutMachine.DoughnutType.Glazed)
    End Sub
```

Visual C#

```
private void mnuRaisedGlazed_Click(object sender,
    System.EventArgs e)
{
    mnuRaisedGlazed.Checked = true;
    mnuRaisedSugar.Checked = false;
    mnuRaisedChocolate.Checked = false;
    myDoughnutMachine.MakeDoughnuts(DoughnutMachine.DoughnutType.Glazed);
}
```

Напишите аналогичные методы для элементов меню *mnu Raised Sugar* и *mnuRaisedChocolate*.

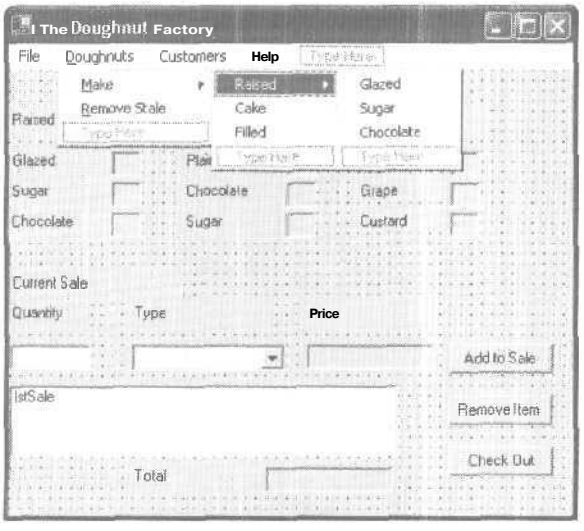


Рис. 3-1. Вид интерфейса с меню

6. Создайте метод, обрабатывающий событие *DoughnutComplete*, которое генерирует объект *myDoughnutMachine*. Обработчик этого события должен увеличить на единицу переменную для учета готовых пончиков данного сорта и записать ее текущее значение а соответствующее текстовое поле. Вот пример реализации обработчика:

Visual Basic .NET

```
Private Sub DoughnutCompleteHandler()
    Select Case myDoughnutMachine.Flavor
        Case DoughnutMachine.DoughnutType.Glazed
            mRaisedGlazed += 1
            txtGlazedRaised.Text = mRaisedGlazed.ToString
        Case DoughnutMachine.DoughnutType.Sugar
            mRaisedSugar += 1
```

```
            txtSugarRaised.Text = mRaisedSugar.ToString
        Case DoughnutMachine.DoughnutType.Chocolate
            mRaisedChocolate += 1
            txtChocolateRaised.Text - roRaisedChocolate.ToString
    End Select
    End Sub
```

Visual C#

```
private void DoughnutCompleteHandler()
{
    switch (myDoughnutMachine.Flavor)
    {
        case DoughnutMachine.DoughnutType.Glazed:
        mRaisedGlazed ++;
        txtGlazedRaised.Text = mRaisedGlazed.ToString();
        break;
    case DoughnutMachine.DoughnutType.Sugar:
        mRaisedSugar ++;
        txtSugarRaised.Text - mRaisedSugar.ToString();
        break;
    case DoughnutMachine.DoughnutType.Chocolate:
        mRaisedChocolate ++;
        txtChocolateRaised.Text - mRaisedChocolate.ToString();
        break;
    }
}
```

7. Добавьте к обработчику события *Form1_Load* следующий код, связывающий обработчик события *DoughnutComplete*, созданный ранее, с событием *myDoughnut-Machine.Doughnut Complete:*

Visual Basic .NET

```
AddHandler myDoughnutMachine.DoughnutComplete, AddressOf _
    DoughnutCompleteHandler
```

Visual C#

```
myDoughnutMachine.DoughnutComplete += new DoughnutMachine.Doughnut-
CompleteDelegate(DoughnutCompleteHandler);
```

8. Добавьте к меню Doughnuts элемент, назовите его **&Stop,** присвойте ему внутреннее имя **mnuStop и** напишите обработчик события *Click* для этого пункта меню, устанавливающий свойство *Enabled* объекта *my Doughnut Machine* в *false*.Вот примерный код этого обработчика:

Visual Basic .NET

```
Private Sub mnuStop_Click(ByVal sender As System.Object, ByVal e _
    As System.EventArgs) Handles mnStop.Click
```

```
    myDoughnutMachine.Enabled = False
End Sub
```

Visual C#

```
private void mnuStop_Click(object sender, System.EventArgs e)
{
    myDoughnutMachine.Enabled = false;
}
```

9. Скомпонуйте и сохраните решение.
10. Нажмите F5, чтобы запустить приложение и проверить, как оно работает.
И. В меню Doughnuts выберите элемент Raised, затем Sugar. Убедитесь, что элемент Sugar помечен флажком. Каждые восемь секунд значение текстового поля Raised Sugar должно увеличиваться на единицу. Выполните те же самые действия в меню Glazed — соответствующее текстовое поле будет увеличиваться каждые 10 секунд; а период обновления поля, соответствующего сорту Chocolate, составит 15 секунд.
12. В меню Doughnuts выберите команду Stop, чтоб закончить «выпечку» пончиков.

Закрепление материала

Приведенные ниже вопросы помогут вам лучше усвоить основные темы данной главы. Если вы не сумеете ответить на вопрос, повторите материал соответствующего занятия. Ответы для самопроверки — в приложении «Вопросы и ответы» в конце книги.

1. Объясните, в каких случаях преобразование типов выполняется неявно, а в каких — явно. В чем опасность явного преобразования типов?
2. Почему вместо значений лучше использовать в коде перечисления и константы?
3. Кратко опишите сходства и различия между массивами и наборами.
4. Чем свойства отличаются от полей? Почему для представления доступа к открытым данным лучше применять свойства, а не поля?
5. Объясните, что такое делегат, и расскажите, как он работает.

ГЛАВА 4

Объектно-ориентированное программирование и полиморфизм

Занятие 1. Введение в объектно-ориентированное программирование 152

Занятие 2. Перегрузка членов 156

Занятие 3. Реализация полиморфизма через интерфейсы 161

Занятие 4. Реализация полиморфизма через наследование 168

В этой главе

Эта глава посвящена основам объектно-ориентированного программирования (ООП). Вы научитесь создавать перегруженные методы и применять принцип полиморфизма через интерфейсы и наследование,

Прежде всего

Для изучения материалов этой главы необходимо знать материал глав 2 и 3,

Занятие 1. Введение в объектно-ориентированное программирование

Программирование в .NET-окружении основано на объектах. Объект — это программная конструкция, инкапсулирующая набор логически связанных данных и методов. Объекты являются автономными элементами, они предоставляют некоторую функциональность другим компонентам среды приложения, изолируя от них свое внутреннее устройство. Объекты создаются на основе шаблонов, называемых классами. Библиотека базовых классов .NET предоставляет набор классов, которые можно применять для создания объектов в собственных приложениях. Кроме того, среда разработки Visual Studio позволяет создавать собственные классы для своих программ. На этом занятии разбираются основные понятия ООП.

Изучив материал этого занятия, вы сможете:

- ✓ описать члены объекта;
- ✓ рассказать о различиях между объектами и классами;
- ✓ дать определение понятию «объектная модель»;
- ✓ дать определение понятиям «абстрагирование», «инкапсуляция» и «полиморфизм».

Продолжительность занятия - около 20 минут.

Объекты, члены и абстрагирование

Объект — это программная конструкция, представляющая некоторую сущность, В нашей повседневной жизни сущностями, или объектами, можно считать автомобили, велосипеды, настольные компьютеры и т.д. Каждый объект обладает определенной функциональностью и свойствами. В приложении объектом может быть форма, элемент управления, такой, как кнопка, соединение с базой данных — в общем, любой элемент. Объект представляет собой завершенную функциональную единицу, которая содержит все данные и предоставляет всю функциональность, необходимую для решения задачи, для которой он предназначен. Представление объектов реального мира при помощи программных объектов называют *абстрагированием* (abstraction).

Классы как шаблоны объектов

При обсуждении классов в главе 1 говорилось, что они представляют собой пользовательские ссылочные типы. Можно сказать, что классы — это «чертежи» объектов. Они определяют все члены объекта к его поведение, а также устанавливают первоначальные значения для данных объекта, если это необходимо. При создании экземпляра класса в памяти создается копия этого класса. Созданный таким образом экземпляр класса называют *объектом.* Экземпляр класса можно создать посредством ключевого слова *New (new),* например, так:

Visual Basic .NET

```
' Объявить переменную типа Widget:
Dim myWidget as Widget
```

```
' Создать экземпляр объекта Widget
' и записать его в переменную myWidget
myWidget = New Widget()
```

Visual C#

```
// Объявить переменную типа Widget
Widget myWidget;
// Создать экземпляр объекта Widget
// и записать его в переменную myWidget
 myWidget = new Widget();
```

При создании экземпляра класса копия данных, определяемых этим классом, записывается в память и присваивается переменной ссылочного типа, Экземпляры класса не зависят друг от друга и являются отдельными программными конструкциями, Как правило, разрешается создавать произвольное число копий класса, которые могут существовать одновременно. Давайте продолжим параллель с реальным миром: если считать автомобиль объектом, чертежи автомобиля представляют собой класс объекта «автомобиль». По чертежу вы сделаете сколько угодно автомобилей. Если один из автомобилей будет собран не так, как все, это никак не повлияет на остальные.

Объекты и члены

Объекты состоят из членов, к которым относят его свойства, поля, методы и события, представляющие данные и функциональность объекта. Поля и свойства содержат данные объекта, методы определяют действия, которые объект способен выполнять, а события представляют уведомления, которые объект может получать или отправлять другим объектам, если происходит нечто интересное или важное.

Вернемся к примеру с объектом *Автомобиль.* Его свойства и поля *Цвет*, *Модель*, *Возраст*, *Расход_топлива* и т.д. представляют данные, описывающие *состояние* объекта, а также методы, например *Нажать_акселератор*, *Переключить_передачу* или *Повернуть_руль.* Методы формируют *поведение* объекта. Событие представляет собой уведомление о важных происшествиях. Для объекта *Автомобиль* необходимо, чтобы объект *Двигатель* уведомлял его о событии *Перегрев,* а взаимодействие с объектом *Дерево* должно вызвать событие *Столкновение.*

Объектные модели

У простых объектов бывает несколько свойств, методов и одно-два события, более сложным объектам требуется больше свойств и методов и даже другие (дочерние) объекты. Сложные объекты способны хранить дочерние объекты и открывать к ним доступ, как к своим членам. Например, элемент управления *TextBox* поддерживает свойство *font,* которое представляет собой объект типа *Font,* Аналогично любой экземпляр класса *Form* содержит и предоставляет набор *Controls*, в который входят все элементы управления, размещенные на форме. Иерархия вложенных (дочерних) объектов, формирующих структуру объекта, определяется его *объектной моделью* (object model).

Объектная модель — это иерархия дочерних объектов главного объекта, к которым он предоставляет доступ. Обратимся к нашему примеру с автомобилем, Автомобиль, который сам по себе является объектом, состоит из ряда дочерних объектов: объекта *Двигатель,* четырех объектов *Колесо,* объекта *Трансмиссия* и т.д. Ком-

поновка дочерних объектов непосредственно определяет работу объекта *Автомобиль*. Например, поведение объектов *Автомобиль*, у которых свойство *Число_цилиндров* дочернего объекта *Двигатель* равно соответственно 4 и 8, будет различным. В свою очередь у дочерних объектов могут быть собственные дочерние объекты. Например, объект *Двигатель* (который является дочерним объектом объекта *Автомобиль*) может иметь несколько дочерних объектов *Свеча_зажигания*.

Инкапсуляция

Инкапсуляция представляет собой один из базовых принципов ООП, суть которого состоит в отделении реализации объекта от его интерфейса. Другими словами, приложение взаимодействует с объектом через его интерфейс, который состоит из открытых свойств и методов. Коль скоро интерфейс объекта остается неизменным, приложение сохраняет способность к взаимодействию с ним, даже если в новой версии объекта его реализация полностью изменится.

Объекты должны взаимодействовать друг с другом только через свои открытые методы и свойства, поэтому объекту требуются все необходимые данные и полный набор методов, необходимых для работы с этими данными. Интерфейс ни в коем случае не должен открывать доступ к внутренним данным объекта, поэтому поля с внутренними данными объекта редко объявляют с модификатором *Public (public)*.

Вернемся к примеру с объектом *Автомобиль,* который может взаимодействовать с объектом *Водитель* через открытый интерфейс, где объявлены методы *Ехать_вперед, Ехать_назад* и *Остановиться* — их достаточно для взаимодействия объектов *Водитель* и *Автомобиль. У* объекта *Автомобиль* может быть дочерний объект *Двигатель,* но это безразлично объекту *Водитель* — ему важно лишь наличие методов, вызывая которые, удается получать соответствующие значения. Так, можно заменить объект *Двигатель,* и взаимодействующий с ним объект *Водитель* не заметит замены, если она не нарушит корректную работу интерфейса.

Полиморфизм

Благодаря полиморфизму, одни и те же открытые интерфейсы удается по-разному реализовать в разных классах. Другими словами, полиморфизм позволяет вызывать методы и свойства объекта независимо от их реализации. Например, объект *Водитель* взаимодействует с объектом *Автомобиль* через одноименный открытый интерфейс. Если другой объект, например *Грузовик* или *Спортивный_автомобиль*, поддерживает этот открытый интерфейс, объект *Водитель* сможет взаимодействовать и с ними, невзирая на различия в реализации интерфейса. Основных подходов к реализации полиморфизма два: через интерфейсы и через наследование.

Реализация полиморфизма через интерфейсы

Интерфейс (interface) — это соглашение, определяющее поведение объекта. Интерфейс определяет список членов класса, но ничего не говорит об их реализации. В объекте допустимо реализовать несколько интерфейсов, а один и тот же интерфейс можно реализовать в разных классах. Любые объекты, в которых реализован некий интерфейс, способны взаимодействовать друг с другом через него. Например, в объекте *Автомобиль*, о котором мы говорим, можно реализовать интерфейс *IDrivable* (по правилам имена интерфейсов должны начинаться с буквы «I») с методами *Ехать_прямо, Ехать_назад* и *Остановиться*. Этот же интерфейс удастся реализовать и в других классах, таких, как *Грузовик*,*Автопогрузчик* или *Катер*. В результате

эти объекты получат возможность взаимодействия с объектом *Водитель*. Объект *Водитель* находится в полом неведении относительно реализации интерфейса, с которым он взаимодействует, ему известен лишь сам интерфейс. Подробнее о реализации полиморфизма через интерфейсы рассказывается на занятии 3.

Реализация полиморфизма через наследование

Наследование позволяет создавать новые классы на основе существующих, при этом в новые классы разрешается включать всю функциональность старых и при необходимости модифицировать их члены. Класс, объявленный на основе другого класса, называется его производным или классом-потомком. У любого класса может быть только один прямой предок — его *базовый класс* (base class). У производного класса окажется тот же набор членов, что и у базового, но при необходимости к производному классу разрешается добавлять дополнительные члены. Можно также изменить реализацию членов, унаследованную от базового класса, переопределив их. Производные классы сохраняют все характеристики своего базового класса и способны взаимодействовать с другими объектами, как экземпляры базового класса. Например, на основе базового класса *Автомобиль* можно объявить производный от него класс *Спортивный_автомобиль*, который, в свою очередь, станет базовым для класса *Спортивный_кабриолет*.Для каждого производного класса разрешены дополнительные члены, но наряду с ними все производные классы сохранят функциональность, определенную в исходном классе *Автомобиль* в неизменном виде. Реализация полиморфизма через наследование более подробно рассматривается на занятии 4.

Резюме

- Представление объектов из реального мира при помощи программных конструкций называется *абстрагированием*. Программные объекты воспроизводят характеристики объектов из реального мира посредством членов и их реализации.
- Классы аналогичны «чертежам» объектов. При создании объекта в памяти создается копия его данных, определенных в классе, и устанавливаются начальные значения для членов-переменных, Класс можно рассматривать как шаблон, на основе которого допустимо создать сколько угодно объектов этого класса,
- Инкапсуляция считается одним из фундаментальных принципов ООП. ОПъект должен содержать все необходимые ему данные и код для манипулирования ими. Данные объекта не должны быть доступными другим объектам. Интерфейсы предоставляют доступ только к свойствам и методам.
- Полиморфизм позволяет по-разному реализовать один и тот же открытый интерфейс в разных объектах. В Visual Basic .NET и Visual C# существуют два основных типа реализации полиморфизма:
  - **через интерфейсы.** Интерфейс определяет список членов, которые должны быть реализованы в объекте, но ничего не говорит об их реализации. В одном объекте можно реализовать несколько несвязанных интерфейсов, а один и тот же интерфейс допустимо реализовать в разных объектах;
  - **через наследование.** Объекты способны наследовать функциональность от других объектов. Производный класс сохраняет полную реализацию своего базового класса; с экземплярами производных классов можно обращаться как с экземплярами их базового класса. При необходимости разрешено реализовать дополнительную функциональность в производных классах.

Занятие 2. **Перегрузка членов**

Благодаря перегрузке удается создать несколько членов с одним и тем же именем, но с разной сигнатурой. Перегрузку чаще всего применяют с методами, C# также позволяет перегружать операторы. На этом занятии вы научитесь создавать перегруженные члены классов.

Изучив материал этого занятия, вы сможете:

- ✓ создавать перегруженные методы;
- ✓ создавать перегруженные операторы (в C#).

Продолжительность занятия - около 15 минут.

Вашему классу может потребоваться член, способный принимать разные наборы параметров, например:

Visual Basic .NET

```
Public Sub Display(ByVal DisplayValue As Integer)
    ' Реализация опущена
End Sub
```

Visual C#

```
public void Display(int DisplayValue)
{
   // Реализация опущена
}
```

Этот метод прекрасно подходит для решения задач, для которых он предназначен. Но что, если этому методу следует передать параметр *duration*? Или передать в параметре *DisplayValue* строку вместо целого числа? В Visual Basic .NET первая проблема решается при помощи необязательных параметров, но пользователям C# эта возможность недоступна. Более того, необязательные параметры бесполезны во второй ситуации, когда метод должен принимать аргументы различных типов.

В таких случаях и применяют перегрузку — создание нескольких методов с одинаковыми именами. Перегруженные методы способны возвращать значения одинаковых типов, разрешается объявлять их с одинаковыми модификаторами доступа, но сигнатуры перегруженных методов должны отличаться. При вызове перегруженного метода исполняющая среда проверяет типы переданных аргументов, сравнивает список аргументов с сигнатурами доступных перегруженных методов, и вызывает тот, чья сигнатура совпала со списком аргументов. Если же ни один из перегруженных методов не принимает указанный набор аргументов, генерируется ошибка.

Чаще всего перегрузку применяют с методами, создание перегруженных методов поддерживает и Visual Basic .NET, и C#; последний *дополнительно* позволяет перегружать операторы, обеспечивая поддержку нестандартной функциональности пользовательских типов данных с операторами.

Создание перегруженных методов

Перегруженные методы создаются так же, как и любые другие. Прежде всего для этого нужно объявить метод с некоторым именем, модификатором доступа, типом возвращаемого значения и списком аргументов. У перегруженного метода должно быть то же имя, что и у существующего метода, но другая сигнатура. Версии перегруженного метода также иногда отличаются уровнями доступа и типом возвращаемого значения (но это не обязательно). Вот пример перегруженного метода:

Visual Basic .NET

```
' Пример перегруженного метода.
Public Sub DisplayMessage(ByVal I As Integer)
   MessageBox.Show(I.ToString())
End Sub
' Следующий метод имеет то же имя, что и предыдущий,
' но отличается от него сигнатурой.
Public Sub DisplayMessage(ByVal S As String)
   MessageBox.Show(S)
End Sub
```

Visual C#

```
// Пример перегруженного метода.
public void DisplayMessage(int I)
{
   MessageBox.Show(I.ToString());
}
// Следующий метод имеет то же имя, что и предыдущий,
// но отличается от него сигнатурой.
public void DisplayMessage(string S)
{
   MessageBox.Show(S);
}
```

В этом примере определены два метода с одинаковыми именами, но разной сигнатурой и реализацией. При вызове метода *Display Message* исполняющая среда проверяет тип переданного аргумента. Если это *String,* вызывается версия метода, которая принимает значение типа *String,* а если — *Integer,* вызывается версия метода, принимающая аргументы соответствующего типа.

Чтобы создать перегруженный метод, выполните следующие действия.

1. Объявите метод, имя которого совпадает с именем одного из существующих методов. Сигнатура нового метода не должна совпадать ни с одной из сигнатур одноименных методов, но для уровня доступа и типа возвращаемого значения это ограничение не действует.
2. Напишите реализацию для нового метода.

Перегрузка операторов в Visual C#

При работе с пользовательскими типами данных порой удобно определять арифметические или логические операторы, а также операторы сравнения, способные работать с этими типами. Рассмотрим следующую структуру:

Visual C#

```
public struct HoursWorked
{
   float RegularHours;
   float OvertimeHours;
}
```

Подобную простую структуру можно использовать в бухгалтерском приложении для учета часов рабочего времени и сверхурочных, но при работе с несколькими экземплярами такой структуры возможны сложности. Представьте, например, что требуется сложить значения двух экземпляров такой структуры. Чтобы решить эту задачу, придется писать новый метод, способный складывать экземпляры этой структуры. Для сложения трех и больше экземпляров потребуется вызвать этот метод несколько раз. К счастью, C# позволяет определять поведение оператора при обработке пользовательских типов данных. Например, чтобы получить возможность суммировать структуру из нашего примера, можно создать перегруженную версию оператора «+», добавив к ней необходимую функциональность.

Перегруженные операторы создают с помощью ключевого слова *operator,* имеющего следующий синтаксис:

Visual C#

```
public static type operator op (Argument1[, Argument2])
{
   // Здесь должна быть реализация.
}
```

где *type* — это тип, с которым работает оператор, он одновременно является типом значения, возвращаемого оператором; *Argument1* и *Argument2* — это аргументы оператора. Любой унарный оператор обрабатывает только один аргумент с типом *type.* Бинарные операторы требуют два аргумента, минимум один из которых должен быть с типом *type. Op* — это, собственно, оператор, например «+», «–», «>», «!=» и т.д. Перегруженные операторы необходимо объявлять с модификаторами *public* (чтобы клиенты, работающие с вашим типом данных, смогли получить к нему доступ) и *static.* Перегруженный оператор следует объявлять внутри определения пользовательского типа данных, для работы с которым он предназначен. Мы рассмотрели пример со структурой, но также можно определить перегруженный оператор, способный работать с классами, — все перегруженные операторы определяются одинаково. Вот пример перегруженного оператора «+», предназначенного для работы с созданной выше структурой *HoursWorked*:

Visual C#

```
public struct HoursWorked
{
   float RegularHours;
   float OvertimeHours;
   // Перегруженный оператор должен быть объявлен в определении класса,
   // для работы с которым он предназначен.
   public static HoursWorked operator + (HoursWorked a, HoursWorked b)
   {
      HoursWorked Result = new HoursWorked();
      Result.RegularHours = a.RegularHours + b.RegularHours;
      Result.OvertimeHours = a.OvertimeHours + b.OvertimeHours;
      return Result;
   }
}
```

Перегруженный оператор используют в программе, как любой другой оператор. Ниже показан пример кода, выполняющего сложение двух экземпляров структуры *Hours Worked*:

Visual C#

```
// Этот пример предполагает, что переменные Sunday и Monday содержат
// экземпляры структуры HoursWorked, инициализированные соответствующими
// значениями.
HoursWorked total = new HoursWorked();
total = Sunday + Monday;
```

Как и в случае с перегруженными методами, допустимо создавать различные реализации перегруженных операторов при условии, что они отличаются сигнатурами, Вернемся к примеру со структурой *HoursWorked,* Допустим, нам потребовалась возможность прибавлять значения типа *Integer* к полю *NormalHours* с помощью оператора «+». Эту задачу удастся решить, создав еще одну версию перегруженного оператора «+»:

Visual C#

```
public struct HoursWorked
{
   float RegularHours;
   float OvertimeHours;
   // Это версия перегруженного оператора, определенная в предыдущем примере.
   public static HoursWorked operator + (HoursWorked a, HoursWorked b)
   {
      HoursWorked Result = new HoursWorked();
      Result.RegularHours = a.RegularHours + b.RegularHours;
      Result.OvertimeHours = a.OvertimeHours + b.OvertimeHours;
      return Result;
   }
```

```
    // Новая версия перегруженного оператора + с новой реализацией.
    public static HoursWorked operator + (HoursWorked a, int b)
    {
        HoursWorked Result = new HoursWorked();
        Result.RegularHours = a.RegularHours + b;
        return Result;
    }
}
```

Чтобы создать перегруженный оператор с помощью Visual C#, выполните следующие действия.

1. Объявите перегруженный оператор внутри определения класса или структуры, для работы с которыми он предназначен, при помоши ключевого слова *operator*. Перегруженные операторы необходимо объявлять с модификаторами *public* и *static*. Если объявляется бинарный оператор, по крайней мере один из его аргументов должен иметь тот же тип, что и возвращаемое значение,
2. Напишите реализацию для оператора.

Резюме

- Благодаря перегрузке удается создать несколько методов с одинаковым именем, но разными реализациями. Перегруженные методы должны отличаться сигнатурами, но могут иметь одинаковые уровни доступа и типы возвращаемых значений. Перегруженные методы объявляются так же, как и обычные,
- В программах на C# можно определять нестандартные версии операторов, предназначенные для работы с пользовательскими типами данных. Перегруженные операторы следует объявлять с модификаторами *public* и *static* и ключевым словом *operator*.

Занятие 3. Реализация полиморфизма через интерфейсы

Интерфейсы определяют поведение класса. Один и тот же интерфейс можно реализовать в самых разных классах, что позволяет им взаимодействовать друг с другом, воплощая концепцию полиморфизма. На этом занятии вы научитесь объявлять и реализовывать интерфейсы, а также узнаете, как объекты взаимодействуют через интерфейсы.

Изучив материал этого занятия, вы сможете:

- ✓ определять интерфейсы;
- ✓ создавать реализацию интерфейса;
- ✓ объяснить, каким образом удается реализовать полиморфизм через интерфейсы.

Продолжительность занятия - около 30 минут.

В данном контексте интерфейс — это некое соглашение. Любой объект, в котором реализован данный интерфейс, гарантированно содержит реализацию для членов, объявленных в этом интерфейсе. Если объект поддерживает некоторый интерфейс, любой другой объект, в котором реализован этот интерфейс, окажется способен взаимодействовать с ним.

Интерфейс определяет только члены, реализованные в объекте. В определении интерфейса ничего не говорится о реализации его членов — оно содержит лишь описание типов параметров и возвращаемых значений. Реализация членов, объявленных в интерфейсе, целиком и полностью ложится на класс, в котором этот интерфейс реализован.

Так, в разных объектах одни и те же члены интерфейса реализуются совершенно по-разному. Рассмотрим интерфейс *IShape*, определяющий единственный метод — *CalculateArea*. При этом класс *Circle*, в котором реализован этот интерфейс, вычисляет площадь фигуры совсем не так, как это делает класс *Square*, в котором также реализован этот интерфейс. Тем не менее объект, которому необходимо взаимодействовать через *IShape*, способен вызвать метод *CalculateArea* из класса *Circle* или *Square* и получить корректный результат.

Определение интерфейсов

Интерфейсы определяются с помощью ключевого слова *Interface*(*interface*), например, так:

Visual Basic .NET

```
Public Interface IDrivable
End Interface
```

Visual C#

```
public interface IDrivable
{
}
```

Этот пример объявляет интерфейс *IDrivable*, но не определяет ни одного члена этого интерфейса. Методы-члены интерфейса определяют с обычной сигнатурой метода, но без модификаторов доступа, таких, как *public, private* и др. Модификатор доступа, заданный для интерфейса, определяет уровень доступа для всех его членов. Таким образом, если интерфейс объявлен с модификатором *Public,* этот модификатор действует для всех его членов. Вот пример определения методов как членов интерфейса:

Visual Basic .NET

```
Public Interface IDrivable
   Sub GoForward(ByVal Speed As Integer)
   SUDHalt()
   Function DistanceTraveled() As Integer
End Interface
```

Visual C#

```
public interface IDrivable
{
   void GoForward(int Speed):
   void Halt():
   int DistanceTraveled():
}
```

Помимо методов, в интерфейсах разрешается определять свойства. На Visual Basic свойства-члены интерфейса необходимо определять с ключевым словом *ReadOnly* или *WriteOnly*. В C# определение свойства включает *метод-получатель* (getter), *метод-установщик* (setter) или оба этих метода. Определение свойства завершается указанием типа значения, возвращаемого свойством. Вот пример определения свойства:

Visual Basic .NET

```
Public Interface IDrivable
' Определения остальных членов опущены.
' Это определение свойства, доступного только для чтения.
   ReadOnly Property FuelLevel() As Integer
End Interface
```

Visual C#

```
public interface IDrivable
{
// Определения остальных членов опущены.
   int FuelLevel
   {
      get;
      // Чтобы сделать это свойство доступным только для чтения,
      // объявите здесь метод set.
   }
}
```

В отличие от свойств, поля не могут быть членами интерфейсов. Это гарантирует, что классы, взаимодействующие через интерфейсы, не получат доступ к внутренним данным объектов.

Интерфейсы также определяют события, генерируемые объектами, в которых этот интерфейс реализован. Любой класс, реализующий некоторый интерфейс, обязан предоставлять реализацию для всех событий-членов этого интерфейса, но объекты, взаимодействующие через этот интерфейс, не обязаны обрабатывать все объявленные в нем события. Visual Basic .NET предоставляет тип-делегат по умолчанию для всех событий, а C# требует явно назначать событиям типы-делегаты. Вот пример определения события, которое является членом интерфейса:

Visual Basic .NET

```
Public Interface IDrivable
    ' Определения остальных членов опущены.
    Event OutOfFuel(ByVal sender As Object, e As System.EventArgs)
End Interface
```

Visual C#

```
public interface IDrivable
{
    // Определения остальных членов опущены.
    event System.EventHandler OutOfFuel;
}
```

Чтобы определить интерфейс, объявите его с помощью ключевого слова *Interface* (Visual Basic) или *interface* (C#) и добавьте к определению интерфейса сигнатуры его членов — методов, свойств и событий.

Интерфейсы как средство достижения полиморфизма

Любой объект, в котором реализован некоторый интерфейс, способен к взаимодействию с любым другим объектом, требующим этот интерфейс, например:

Visual Basic .NET

```
Public Sub GoSomewhere(ByVal V As IDrivable)
    ' Реализация опущена
End Sub
```

Visual C#

```
public void GoSomewhere(IDrivable v)
{
    // Реализация опущена
}
```

Объявленному здесь методу необходимо, чтобы в его аргументе был реализован интерфейс *IDrivable*. Следовательно, этому методу можно передать любой объект, где реализован этот интерфейс, попутно этот объект неявно преобразуется в тип соответствующего интерфейса. Объектам, взаимодействующим через реализованный в них интерфейс, доступны только члены этого интерфейса.

Кроме того, разрешается явно преобразовывать объекты, в которых реализован некоторый интерфейс, к типу этого интерфейса. Следующий пример демонстрирует преобразование объекта *Truck* к типу интерфейса *IDrivable* (заметьте: чтобы это сработало, в объекте *Truck* необходимо реализовать интерфейс *IDrivable);*

Visual Basic .NET

```
Dim myTruck As New Truck()
Dim myVehicle As IDrivable
' Преобразование объекта myTruck в тип интерфейса IDrivable
myVehicle = CType(IDrivable, myTruck)
```

Visual C#

```
Truck myTruck = new Truck();
IDrivablemyVehicle;
// Преобразование объекта myTruck в тип интерфейса IDrivable
myVehicle = (IDrivable)myTruck;
```

Реализация интерфейсов

Чтобы в программе на Visual Basic реализовать интерфейс в классе или структуре, следует воспользоваться ключевым словом *Implements*, а в Visual C# для этой цели применяется двоеточие, вот пример:

Visual Basic .NET

```
Public Class Truck
    Implements IDrivable
    ' Реализация опущена
End Class
```

Visual C#

```
public class Truck : IDrivable
{
   // Реализация опущена
}
```

В классах реализуется несколько интерфейсов. Чтобы объявить такой класс, укажите все необходимые интерфейсы через запятую, как показано ниже.

Visual Basic .NET

```
Public Class Truck
   Implements IDrivable, IFuelBurning, ICargoCarrying
   ' Реализация опущена
End Class
```

Visual C#

```
public class Truck : IDrivable, IFuelBurning, ICargoCarrying
{
```

```
   // Реализация опущена
}
```

Если в классе или структуре реализован некоторый интерфейс, необходимо предоставить реализацию для всех его членов, то же правило действует, если реализовано несколько интерфейсов.

Реализация членов интерфейса на Visual C#

На Visual C# члены интерфейсов, объявленных в классах и структурах, реализуют следующим образом. Для этого определяют член с именем, совпадающим с именем члена интерфейса, причем этот член класса объявляется с тем же уровнем доступа, с которым объявлен интерфейс. Вот пример реализации метода как члена интерфейса:

Visual C#

```
public interface IDrivable
{
   void GoForward(int Speed);
}
public class Truck : IDrivable
{
   public void GoForward(int Speed)
   {
      // Реализация опущена
   }
}
```

Члены интерфейса, реализованные таким способом, становятся доступны как интерфейсу, так и самому классу. В результате эти члены можно вызывать после преобразования объекта, где интерфейс реализован, в его собственный тип либо в тип этого интерфейса.

Кроме того, допустимо реализовать интерфейс в классе явно. Члены, реализованные подобным образом, доступны только после преобразования объекта в тип интерфейса. Чтобы реализовать член интерфейса явно, следует объявить в классе одноименный член, указав полное имя интерфейса. Вот пример явно реализованного метода *GoForward*, являющегося членом интерфейса *IDrivable:*

Visual C#

```
public class Truck : IDrivable
{
   void IDrivable.GoForward(int Speed)
   {
      // Реализация опущена
   }
}
```

Обратите внимание, что у этого члена нет модификатора доступа. Поскольку этот член интерфейса реализован явно, его уровень доступа определяется модификатором, заданным для интерфейса.

Чтобы реализовать интерфейс в программе на C#, необходимо выполнить следующие действия.

1. Объявите класс с нужным интерфейсом при помощи двоеточия (;). Так можно указать несколько интерфейсов, разделив их имена запятыми.
2. Напишите реализацию для всех членов интерфейса:
   - чтобы член оказался доступен как через класс, так и через интерфейс, объявите этот член с именем, уровнем доступа и сигнатурой, идентичным таковым членам интерфейса;
   - чтобы сделать член доступным только через интерфейс, объявите его при помощи полного имени интерфейса, указывать модификатор доступа при этом не обязательно.

Реализация членов интерфейса на Visual Basic .NET

В программе на Visual Basic .NET ключевое слово *Implements* позволяет указать, что данный член класса является реализацией члена интерфейса. Сигнатура члена класса, реализующего член некоторого интерфейса, должна быть идентична сигнатуре реализуемого им члена интерфейса, исключение составляет модификатор доступа. Следующий пример демонстрирует реализацию метода, являющегося членом интерфейса:

Visual Basic .NET

```
Public Interface IDrivable
   Sub GoForward(ByVal Speed As Integer)
End Interface
Public Class Truck
   Implements IDrivable
   Public Sub GoForward(ByVal Speed As Integer) Implements _
      IDrivable.GoForward
   ' Реализация опущена
   End Sub
End Class
```

Член класса, реализующий член интерфейса, вовсе не обязан иметь то же имя, что и член интерфейса. Например, вполне допустима следующая реализация метода *GoForward* из интерфейса *IDrivable*:

Visual Basic .NET

```
Public Sub Move(ByVal Speed As Integer) Implements IDrivable.GoForward
' Реализация опущена
End Sub
```

Любой вызов метода *GoForward*, объявленного в интерфейсе *IDrivable*, в этом примере разрешается на метод *Move* из класса, реализующего метод интерфейса.

Кроме того, методу класса, реализующему метод интерфейса, можно назначить уровень доступа, отличный от уровня доступа к интерфейсу. Например, разрешается реализовать метод интерфейса, объявленного с уровнем доступа *Public,* через метод класса, объявленный с уровнем доступа *Private.* В этом случае при доступе к такому методу через интерфейс действует уровень доступа *Public,* а при доступе через класс — уровень доступа *Private.*

Чтобы реализовать интерфейс на Visual Basic .NET, необходимо выполнить следующие действия.

1. Объявите класс с необходимым интерфейсом, указав его при помощи ключевого слова *Implements*. В классе можно реализовать несколько интерфейсов.
2. Воспользуйтесь ключевым словом *Implements*, чтобы создать реализацию для всех членов интерфейса, которые предполагается реализовать в классе.

Резюме

- Интерфейс определяет поведение объекта и содержит описание поддерживаемых им членов, их параметров, а также типов возвращаемых ими значений. Любой объект, где реализован некоторый интерфейс, способен взаимодействовать с любым другим объектом, которому необходим этот интерфейс. Интерфейсы реализуются как в классах, так и в структурах; в тех и других допустимо реализовать несколько интерфейсов.
- За реализацию членов интерфейса отвечает класс или структура, в которой этот член реализован.
- Объявить, что в классе или структуре реализован некоторый интерфейс, позволят ключевое слово *Implements* (в Visual Basic .NET) или двоеточие (в C#). Класс или структура, реализующая некоторый интерфейс, должна обеспечить реализацию всех членов, объявленных в этом интерфейсе.
- В C# члены интерфейса реализуют двумя способами:
  - реализацией члена с именем, сигнатурой и уровнем доступа, идентичным таковым соответствующего члена интерфейса. Такие члены доступны как через имя класса, в котором реализован этот интерфейс, так и через имя интерфейса;
  - явной реализацией члена интерфейса с указанием его полного имени. Такие члены доступны только через имя интерфейса.
- В Visual Basic .NET члены интерфейса реализуют посредством членов класса или структур с сигнатурой, идентичной сигнатуре соответствующего члена интерфейса. При этом ключевое слово *Implements* позволяет увязать член интерфейса с соответствующим членом класса. Имя и уровень доступа члена класса могут отличаться от таковых члена интерфейса.

Занятие 4. Реализация полиморфизма через наследование

Наследование позволяет объявить новый класс на основе существующего, передав новому классу все члены и функциональность существующего класса. Так удается создавать классы, реализующие базовый набор возможностей, а затем использовать их для объявления производных классов для исполнения различных, но связанных функций. На этом занятии вы научитесь применять наследование для создания производных классов на основе базовых, создавать новую реализацию для членов базовых классов, а также объявлять абстрактные базовые классы,

Изучив материал этого занятия, вы сможете:

- ✓ расширять возможности базового класса;
- ✓ переопределять и замещать члены базовых классов;
- ✓ дать определение понятию абстрактного класса;
- ✓ создавать абстрактные члены.

Продолжительность занятия - около 60 минут.

Наследование

Наследование позволяет создавать классы, реализующие общий набор возможностей. При этом специализированные классы, которые также называют *производными,* являются *потомками* одного общего класса, называемого *базовым.* Говорят, что производные классы *расширяют* возможности базового. Базовый класс инкапсулирует общую функциональность, которой будут обладать все производные классы, а производные классы поддерживают дополнительные возможности, уникальные для каждого производного класса. Рассмотрим класс *Грузовик,* который представляет всю базовую функциональность автомобиля (например, методы для движения вперед, назад и поворота) и инкапсулирует все необходимые объекты. На его основе можно создать производный класс *Пикап,* который унаследует от него всю базовую функциональность. Далее можно реализовать дополнительные возможности, свойственные только классу *Пикап,* например добавить свойство *Тип_груза.* В свою очередь, класс *Пикап* можно назначит базовым, например, для класса *Полноприводной_пикап.*

Полиморфизм и производные классы

Производный класс, как правило, можно отождествлять с его базовым классом. Например, любой объект класса *Полноприводной_пикап* также является объектом класса *Пикап,* а все объекты класса *Пикап* представляют собой объекты класса *Грузовик.* Полиморфизм при наследовании заключается в том, что любой экземпляр производного класса способен выполнять функции своего базового класса. Так, методу, требующему объект *Грузовик,* допустимо передать объект *Пикап.* Любой производный класс можно неявно преобразовать в объект своего базового класса, при этом все члены, реализованные в производном классе, окажутся недоступными, доступными останутся только члены базового класса.

Создание производных классов

Производные классы объявляют при помощи ключевого слова *Inherits (в* Visual Basic .NET) или двоеточия (в C#), Вот пример объявления класса *PickupTruck*, производного от класса *Truck:*

Visual Basic .NET

```
Public Class PickupTruck
   Inherits Truck
   ' Реализация опущена
End Class
```

Visual C#

```
public class PickupTruck : Truck
{
   // Реализация опущена
}
```

У производного класса только один предок (базовый класс), но дополнительно в нем разрешается реализовать один или несколько интерфейсов. Если вы объявляете производный класс, в котором реализовано несколько интерфейсов, следует сразу после ключевого слова *Inherits* поместить ключевое слово *Implements* (Visual Basic .NET) либо перечислить все интерфейсы после двоеточия, следующего за именем базового класса (C#),

Visual Basic .NET

```
Public Class FourWheelDrivePickupTruck
   Inherits PickupTruck
   Implements IFourWheelDrive
   ' Реализация опущена
End Class
```

Visual C#

```
public class FourWheelDrivePickupTruck : PickupTruck, IFourWheelDrive
{
   // Реализация опущена
}
```

Объявив производный класс, можно добавить к нему дополнительные члены, реализующие нестандартные возможности.

▶ **Как объявить производный класс**

* в **Visual Basic .NET** следует указать базовый класс при помощи ключевого слова *Inherits*;
* в **C#** указать базовый класс после двоеточия в той же строке.

Создание запечатанных классов

Иногда требуются классы, которые не способны играть роль базовых. Например, специализированные классы, предназначенные для использования исключительно

компонентами вашего приложения. Они не пригодятся другим программистам, поскольку не способны дать своим потомкам никакой полезной функциональности. Такие классы можно пометить ключевым словом *NotInheritable* (Visual Basic .NET) или *sealed* (C#):

Visual Basic .NET

```
Public NotInheritable Class AClass
    ' Реализация опущена
End Class
```

Visual C#

```
public sealed class AClass
{
   // Реализация опущена
}
```

Наследуемые члену

Как говорилось ранее, производный класс обладает всей функциональностью базового. В производных классах разрешается не только добавлять новые члены, но и изменять реализацию унаследованных членов для решения поставленных задач. *Переопределив* (override) члены, унаследованные от базового класса, удается изменять их реализацию; Visual Basic .NET также позволяет *замещать* (shadow) члены, унаследованные от базового класса. Их можно скрывать, предоставляя взамен новый член с тем же именем, но с совершенно *иной* сигнатурой, уровнем доступа и даже типом. C# позволяет вам *скрывать* (hide) члены, унаследованные от базового класса, и реализовать взамен член с идентичным именем и сигнатурой, но отличающийся другими характеристиками. Переопределение, замещение и сокрытие членов более подробно рассматривается в следующих разделах.

Переопределение членов, унаследованных от базового класса

Чтобы изменить реализацию членов базового класса, следует переопределить их. Этот механизм позволяет заменить член, унаследованный от базового класса, членом с тем же именем, но с другой реализацией. Например, класс *Автомобиль* имеет метод *Ехать_прямо*. У производного класса *Спортивный_автомобиль* реализация этого метода может отличаться. Переопределять разрешено только свойства и методы, унаследованные от базового класса, — переменные и события переопределять нельзя.

Для объявления другой реализации члена применяют ключевое слово *Overrides* (Visual Basic .NET) или *override* (C#). В новой реализации члена сигнатура, тип возвращаемого значения и уровень доступа должны совпадать с таковыми переопределяемого члена, например:

Visual Basic .NET

```
' Этот пример предполагает наличие
' у базового класса метода GoForward.
Public Class SportsCar
```

```
    Inherits Car
    ' Объявить переопределяемый метод
    Public Overrides Sub GoForward(ByVal Speed As Integer)
        ' Реализация опущена
    End Sub
End Class
```

Visual C#

```
// Этот пример предполагает наличие
// у базового класса метода GoForward.
public class SportsCar : Car
{
    public override void GoForward(int Speed)
    {
        // Реализация опущена
    }
};
```

При вызове переопределенного члена вместо члена, унаследованного от базового класса, вызывается новая реализация члена независимо от контекста вызова. Например, если после преобразования экземпляра производного класса в тип его базового класса вызвать переопределенный метод, все равно исполняется новая реализация этого метода. Вызываемый член определяется объектом, а не типом переменной.

Чтобы член базового класса удалось переопределить, он должен быть помечен ключевым словом ***Overridable*** (Visual Basic .NET) или *virtual* (C#), в противном случае члены считаются фиксированными и их переопределить нельзя. Вот пример объявления метода с помощью ключевого слова *Overridable* (*virtual*):

Visual Basic .NET

```
Public Overridable Sub OverrideMe()
    ' Реализация опущена
End Sub
```

Visual C#

```
public virtual void OverrideMe()
{
    // Реализация опущена
}
```

▶ **Как переопределить член базового класса**

1. Убедитесь, что член, который вы пытаетесь переопределить, объявлен с ключевым словом *Overridable (virtual),* если это не так, этот член не удастся переопределить, поэтому его придется заместить (скрыть). Как это сделать, рассказано далее.
2. Напишите новую реализацию члена при помощи ключевого слова *Overrides (override).*

Замещение членов базового класса в Visual Basic .NET

Из предыдущего раздела вы узнали, как заменить реализацию члена, унаследованного от базового класса. Можно также полностью скрыть реализацию члена базового класса, заменив его совершенно новым членом с другим уровнем доступа, сигнатурой или членом другого типа. Этот прием называется *замещением* (shadowing). В Visual Basic .NET для замещения применяется ключевое слово *Shadows*.

Visual Basic .NET

```
' Это базовый класс
Public Class MyBaseClass
   Public Function MyMethod(ByVal I As Integer) As String
      ' Реализация опущена
   End Function
End Class
' Это производный класс
Public Class MyInheritedClass
   Inherits MyBaseClass
   ' Эта функция замещает метод MyMethod, унаследованный от базового класса.
   ' Обратите внимание, что у нее совершенно другая сигнатура,
   ' уровень доступа и тип возвращаемого значения.
   Friend Shadows Function MyMethod(ByVal S As String) As Integer
      ' Реализация опущена
   End Function
End Class
```

Сокрытие членов базового класса в Visual C#

Если вы пишете на C#, у вас есть возможность заменить член, унаследованный от базового класса, другим членом с совершенно иной реализацией. Этот прием называется *сокрытием* (hiding). Тип и сигнатура нового члена должны быть теми же, что и у замещаемого им скрытого члена, но уровень доступа, тип возвращаемого значения и реализация могут отличаться. Если имя нового метода совпадает с именем одного из существующих, но сигнатуры этих методов отличаются, новый метод рассматривается как перегруженная версия существующего метода, при этом последний останется доступным. Чтобы скрыть член, унаследованный от базового класса, необходимо воспользоваться ключевым словом *new*, например, так:

Visual C#

```
// Это базовый класс
public class MyBaseClass
{
   public string MyMethod(int I)
   {
      // Реализация опущена
   }
}
// Этот производный класс
```

```
public class MyInheritedClass : MyBaseClass
{
   // Эта функция замещает метод MyMethod из базового
   // класса. Обратите внимание, что у нее та же сигнатура,
   // но другой уровень доступа и тип возвращаемого значения.
   internal new int MyMethod(int I)
   {
      // Реализация опущена
   }
}
```

Обеспечение совместимости с замещенными или скрытыми членами

Реализация замещенного или скрытого члена базового класса становится недоступной, ее заменяет новая реализация, возможно, обладающая совершенно иными характеристиками, что иногда серьезно влияет на взаимодействие с другими объектами. Объект, вызывающий функцию *MyMethod,* объявленную в предыдущем примере, ожидает значение типа *String,* а если функция вернет *Integer,* возможно возникновение ошибки. Поэтому следует очень осторожно применять замещение и сокрытие членов и прибегать к подобным приемам, только когда вы полностью уверены, что они не нарушат совместимость.

В некоторых случаях получить доступ к реализации замешенных или скрытых членов базового класса все же удается. Откуда будет вызван член — из базового класса или из производного — зависит от типа переменной, а не от объекта, как в случае переопределенного члена. Рассмотрим следующий пример:

Visual Basic .NET

```
' В этом примере используются классы MyBaseClass и MylnheritedClass,
' определенные в предыдущем примере кода.
Dim X As New MyInheritedClass()
Dim Y As MyBaseClass
' Переменные X и Y теперь ссылаются на один и тот же объект,
' но типы этих переменных отличаются.
Y = X
' Тип переменной X - MylnheritedClass, поэтому в
' следующей строке будет вызван член из производного класса.
X.MyMethod("A String")
' Тип переменной Y - MyBaseClass, поэтому в следующей строке
' будет вызвана реализация из базового класса.
Y.MyMethod(42)
```

Visual C#

```
// В этом примере используются классы MyBaseClass и MylnheritedClass,
// определенные в предыдущем примере кода.
MylnheritedClass X - new MyInheritedClass();
MyBaseClass Y;
// Переменные X и Y теперь ссылаются на один и тот же объект,
// но типы этих переменных отличаются,
```

```
Y = X;
// Тип переменной X - MyInheritedClass, поэтому
// в следующей строке будет вызван член из производного класса.
X.MyMethod(42);
// Тип переменной Y -- MyBaseClass, поэтому в следующей строке
// будет вызвана реализация из базового класса.
Y.MyMethod(42);
```

Как вы видите, именно тип переменной определяет, будет ли вызван замещающий член из производного класса, или исходный член из базового класса. Это позволяет модифицировать реализацию члена, не нарушая принципа полиморфизма.

При объявлении производных классов новая реализация для замещенных или скрытых членов не наследуется, производному классу передается реализация соответствующих членов из базового класса.

▶ **Как заместить (скрыть) член базового класса**

- **Visual Basic .NET**: объявите новый член с ключевым словом *Shadows* и напишите для него реализацию. У нового члена возможна другая сигнатура, уровень доступа и тип возвращаемого значения, это может быть даже член другого типа.
- **C#**: объявите новый член с ключевым словом *new* и напишите для него реализацию. У нового члена должны быть такие же сигнатура и тип, что и у скрытого, но уровень доступа и тип возвращаемого значения могут отличаться.

Получение доступа к членам базового класса

При объявлении переопределенных или скрытых членов иногда требуется доступ к реализации данного члена в базовом классе. Его получают через ключевые слова *MyBase* (Visual Basic .NET) или *base* (C#), предоставляющие ссылку на реализацию члена в базовом классе. Эти ключевые слова позволяют вызывать члены из базового класса следующим образом:

Visual Basic .NET

```
' Этот пример демонстрирует вызов реализации базового класса
' из переопределенного метода MyMethod.
Public Overrides Sub MyMethod()
   MyBase.MyMethod()
   ' Реализация опущена
End Sub
```

Visual C#

```
// Этот пример демонстрирует вызов реализации базового класса
// из переопределенного метода MyMethod.
public override void MyMethod()
{
   base.MyMethod();
   // Реализация опущена
}
```

Защищенные члены

С уровнями доступа к членам классов вы познакомились в главе 1: члены, объявленные с уровнем доступа *Public* (*public*), доступны из любой части приложения, в том числе из внешних классов; члены с уровнем доступа *Friend* или *internal* доступны только членам из локальной сборки, но не из внешних сборок; члены с уровнем доступа *Private* (*private*) доступны только в пределах класса и не доступны никаким другим классам, в том числе и производным от класса, в котором объявлен этот член. Остальные два уровня доступа, *Protected (protected)* и *Protected Friend* (*protected internal*), подробно не обсуждались в главе 1.

Области видимости членов классов, объявленных с ключевыми словами *Protected (protected)* и *Private (private),* совпадают за исключением того, что в первом случае члены доступны и производным классам. Рассмотрим следующий пример;

Visual Basic .NET

```
' В базовом классе определены два метода:
' с уровнями доступа private и protected.
Public Class BaseClass
    ' Метод с уровнем доступа private нельзя вызывать
    ' из производных классов...
    Private Sub Method1()
        ' Реализация опущена
    End Sub
    ' ...а метод с уровнем доступа protected
    ' из производных классов вызывать можно.
    Protected Sub Method2()
        ' Реализация опущена
    End Sub
End Class
' Класс, производный от BaseClass
Public Class InheritedClass
    Inherits BaseClass
    Public Sub Demo()
        ' Такой вызов допустим, поскольку уровень доступа protected
        ' не запрещает доступ к члену, объявленному в BaseClass,
        Me.Method2()
        ' а этот вызов приведет к ошибке компиляции, поскольку он
        ' обращается к члену базового класса, объявленному
        ' с уровнем доступа private.
        Me.Method1()
    End Sub
End Class
```

Visual C#

```
// В базовом классе определены два метода:
// с уровнями доступа private и protected.
public class BaseClass
```

```
{
   // Метод с уровнем доступа private нельзя вызывать
   // из производных классов...
   private void Method1()
   {
      // Реализация опущена
   }
   // ...а метод с уровнем доступа protected
   // из производных классов вызывать можно.
   protected void Method2()
   {
      // Реализация опущена
   }
}
// Класс, производный от BaseClass
public class InheritedClass : BaseClass
{
   public void Demo()
   {
      // Такой вызов допустим, поскольку уровень доступа protected
      // не запрещает доступ к члену, объявленному в BaseClass,
      this.Method2();
      // а этот вызов приведет к ошибке компиляции, поскольку он
      // обращается к члену базового класса, объявленному
      // с уровнем доступа private.
      this.Method1();
   }
}
```

Здесь иллюстрируется различие закрытых и защищенных членов. Метод *Demo* попытается вызвать методы *Method!* и *Method2* при помощи ссылки на экземпляр класса *InheritedClass* (через ключевое слово *Me* [*this*]). Поскольку оба этих метода определены в базовом классе, возможность их вызова определяется уровнями доступа, с которыми они объявлены. Вызов метода *Method2* завершается успешно, так как он объявлен с модификатором *Protected (protected),* а метод *Method1,* объявленный с модификатором *Private (private),* производным классам не доступен.

Уровень доступа *Protected Friend (protected internal)* представляет собой «гибрид» модификаторов *Protected (protected)* и *Friend (internal).* Поэтому член класса, объявленный с модификатором *Protected Friend (protected internal),* доступен членам этого класса и других классов, расположенных в той же сборке с этим классом, а также членам своих производных классов.

Абстрактные классы и члены

При создании компонентов иногда требуются классы, которые предоставляют некоторый набор возможностей, используемых в неизменном виде, в то время как за реализацию остальных членов таких классов отвечают их производные классы. Такими возможностями обладают *абстрактные* (abstract) классы, которые способны только выполнять роль базовых классов.

Абстрактные классы напоминают интерфейсы, но у них также много общего с обычными классами. Создать экземпляр абстрактного класса невозможно, он годится лишь для объявления производных классов. Абстрактный класс предоставляет полную реализацию класса, ее часть или вовсе не предоставляет никакой реализации. И интерфейсы, и абстрактные классы содержат описание членов, которые следует реализовать в производных классах, но, в отличие от интерфейсов, только один абстрактный класс может быть предком производного класса. Абстрактные классы предоставляют лишь полностью реализованные члены (этим они похожи на обычные классы) и члены, за реализацию которых отвечают производные классы (это отличие абстрактных классов).

Создание абстрактных классов

Абстрактные классы объявляют с помощью ключевого слова *MustInherit* (Visual Basic .NET) или *abstract* (в C#), например:

Visual Basic .NET

```
Public MustInherit Class AbstractClass
    ' Реализация опущена
End Class
```

Visual C#

```
public abstract class AbstractClass
{
    // Реализация опущена
}
```

Создание абстрактных членов

У абстрактного класса могут быть различные члены, как у обычного класса. Их объявляют с ключевым словом *Overridable* (*virtual*) — в этом случае в производных классах они реализуются по-разному; в противном случае реализация члена считается фиксированной и в неизменном виде передается всем производным классам.

Кроме того, в абстрактных классах допустимо объявлять абстрактные члены. Процедура их объявления во многом напоминает объявление члена интерфейса. При этом указывают только тип члена, уровень доступа, необходимые параметры и тип возвращаемого значения. Объявление абстрактного члена не содержит никаких деталей его реализации — только его интерфейс.

Абстрактные члены объявляют при помощи ключевого слова *MustOverride* (Visual Basic .NET) или *abstract* (C#). Определять абстрактные члены можно только в абстрактных классах, в противном случае возникает ошибка компиляции. Вот пример абстрактного класса с тремя абстрактными членами:

Visual Basic .NET

```
Public MustInherit Class Car
    Public MustOverride Sub GoForward(ByVal I As Integer)
    Public MustOverride Function CheckSpeed() As Integer
    Public MustOverride Property Color() As String
End Class
```

Visual C#

```
public abstract class Car
{
   public abstract void GoForward(int I);
   public abstract int CheckSpeed();
   public abstract string Color
   {
      get;
      set;
   }
}
```

Обратите внимание, что при определении абстрактного свойства на Visual C# необходимо объявить метод-получатель, метод-установщик или оба. За реализацию этих методов отвечает производный класс. В Visual Basic .NET предполагается, что свойство доступно для чтения и записи, если не указано ключевое слово *Readonly* или *WriteOnly*.

Наследование от абстрактного класса

Классы, производные от абстрактных, если они сами таковыми не являются, должны предоставлять реализацию для всех абстрактных членов, объявленных в классе-предке, Чтобы предоставить реализацию для абстрактного члена, необходимо переопределить его как член любого другого класса. Ниже показан пример реализации абстрактного класса, объявленного в предыдущем примере:

Visual Basic .NET

```
Public Class MyCar
   Inherits Car
   Public Overrides Sub GoForward(ByVal I As Integer)
      ' Сюда нужно поместить код реализации этого метода,
   End Sub
   Public Overrides Function CheckSpeed() As Integer
      ' а сюда - реализацию этого метода.
   End Function
   Public Overrides Property Color() As String
      Get
         ' Необходимо написать реализацию для получателя
         ' и установщика свойства
      End Get
      Set(ByVal Value As String)
         ' Сюда нужно поместить код реализации установщика
      End Set
   End Property
End Class
```

Visual C#

```
public class MyCar : Car
{
   public override void GoForward(int I)
   {
      // Сюда нужно поместить код реализации этого метода,
   }
   public override int CheckSpeed()
   {
      // а сюда - реализацию этого метода.
   }
   public override string Color
   {
      get
      {
         // Необходимо написать реализацию для получателя
         // и установщика свойства
      }
      set
      {
         // Сюда нужно поместить код реализации установщика
      }
   }
}
```

Разрешено также создавать новые абстрактные классы, объявляя их на основе существующих. В этом случае производный абстрактный класс может (но не обязан) предоставлять реализацию для абстрактных членов, определенных в базовом классе.

▶ **Как объявить абстрактный класс**

1. Объявите абстрактный класс при помощи ключевого слова *MustInherit* (Visual Basic .NET) или *abstract* (C#).
2. Объявите все необходимые абстрактные члены при помощи ключевых слов *MustOverride* (Visual Basic .NET) или *abstract* (C#).
3. Напишите реализацию для неабстрактных членов, которые должны быть в вашем классе.

Резюме

- Наследование позволяет создавать классы, в которых функциональность существующего класса объединяется с возможностями новых классов. Производные классы содержат все члены своего базового класса и благодаря полиморфизму способны выполнять функции экземпляров базового класса.
- Для поддержки нестандартной функциональности к производному классу добавляют дополнительные члены либо заменяют в них реализацию членов, унаследованных от базового класса. Для этого предусмотрено два способа:
  - переопределение членов;
  - замещение (Visual Basic .NET) или сокрытие (C#) членов.
- Доступ к реализации члена в базовом классе обеспечивается через ключевое слово *MyBase* (Visual Basic .NET) или *base* (C#). Чтобы сделать член, объявленный в базовом классе, доступным для производного класса, объявите его с модификатором *Protected* (*protected*).
- Абстрактный класс — это своего рода интерфейс, предоставляющий лишь частичную реализацию класса или вовсе не предоставляющий ее. Не разрешается создавать экземпляры абстрактных классов, они служат исключительно для объявления производных классов. Чтобы предоставить реализацию для абстрактного члена, последний необходимо переопределить в производном классе.

Практикум 4. Применение производных классов

Задача этого практикума — создание производных классов и их применение в приложении Doughnut Factory. Вы создадите класс строго типизированного набора путем расширения возможностей базового класса набора и задействуете его в своем приложении. Далее вы объявите на основе класса *Doughnut* классы *CakeDoughnut* и *FilledDoughnut* и внесете в приложение соответствующие изменения, чтобы оно смогло использовать новые классы. Решение заданий этого практикума вы найдете в каталоге \Labs\Ch04\Solution на прилагаемом к книге компакт-диске.

Прежде **всего**

Для этого практикума вам потребуются выполненные задания практикума 3, их можно также загрузить с прилагаемого компакт-диска.

Продолжительность практикума - **около 60** минут.

Упражнение 1. Создание класса строго типизированного набора на основе класса CollectionBase

Сейчас вы создадите класс набора, который воспринимает и возвращает исключительно экземпляры класса *Doughnut*, объявив производный класс на основе класса *CollectionBase* из .NET Framework. Вы напишете реализацию для методов *Add* и *Remove*, а также для свойства *Item*, включив в нее код, обеспечивающий добавление к набору и извлечение из него только объектов типа *Doughnut*. Вы также модифицируете код приложения Doughnut Factory, чтобы оно могло использовать новый класс набора.

▶ Создание класса строго типизированного набора

1. В меню Project выберите команду Add Class — откроется окно Add New Item. Назовите класс **DoughnutCollection.vb** или **DoughnutCollection.cs** и щелкните OK — откроется окно редактора кода с заготовкой нового класса.
2. Сделайте новый класс потомком *System.Collections.CollectionBase*, добавив к нему следующий код:

Visual Basic .NET

```
Public Class DoughnutCollection
   Inherits System.Collections.CollectionBase
End Class
```

Visual C#

```
public class DoughnutCollection : System.Collections.CollectionBase
{
}
```

3. Создайте метод *Add*, который будет добавлять элементы к набору. Этот метод должен принимать объекты *Doughnut* в качестве параметра и вызывать метод *Add* внутреннего объекта *List*.

Visual Basic .NET

```
Public Sub Add(ByVal aDoughnut As DoughnutMachine.Doughnut)
   List.Add(aDoughnut)
End Sub
```

Visual C#

```
public void Add(DoughnutMachine.Doughnut aDoughnut)
{
   List.Add(aDoughnut);
}
```

4. Напишите доступный только для чтения индексатор (C#) или свойство по умолчанию с именем *Item* (Visual Basic), возвращающее объект *Doughnut* с заданным индексом. Заметьте: элементы объекта *List* хранятся, как объекты типа *Object*, поэтому вам потребуется явно преобразовать привести элемент набора к типу *Doughnut*, прежде чем возвращать его. Это можно сделать так:

Visual Basic .NET

```
Default Public Readonly Property Item(ByVal Index As Integer)
   Get
      Return CType(List.Item(Index), DoughnutMachine.Doughnut)
   End Get
End Property
```

Visual C#

```
public DoughnutMachine.Doughnut this[int index]
{
   get
   {
      return (DoughnutMachine.Doughnut)List[index];
   }
}
```

5. Напишите метод, который будет удалять из набора объект *Doughnut*. Этот метод должен принимать объект *Doughnut*, его следует реализовать как оболочку для метода *List.Remove*. Вот пример:

Visual Basic .NET

```
Public Sub Remove(ByVal aDoughnut As DoughnutMachine.Doughnut)
   ' Проверить, является ли заданный объект Doughnut членом набора
   If List.Contains(aDoughnut) = True Then
      ' Удалить заданный объект Doughnut
      List.Remove(aDoughnut)
```

```
    End If
End Sub
```

Visual C#

```
public void Remove(DoughnutMachine.Doughnut aDoughnut)
{
    if (List.Contains(aDoughnut) == true)
    {
        List.Remove(aDoughnut);
    }
}
```

6. Откройте код класса *DoughnutMachine*, найдите строку, в которой объявляется закрытая переменная *mDoughnuts,* и измените ее так, чтобы в ней можно было хранить экземпляр класса *DoughnutCollection:*

Visual Basic .NET

```
Private mDoughnuts As New DoughnutCollection()
```

Visual C#

```
private DoughnutCollection mDoughnuts = new DoughnutCollection();
```

7. Измените свойство *Doughnuts,* чтобы оно получило возможность взаимодействовать с новым объектом *DoughnutCollection,* например, так:

Visual Basic .NET

```
Default Public Property Doughnuts(ByVal Index As Integer) As Doughnut
    Get
        Return mDoughnuts(Index)
    End Get
    Set(ByVal Value As Doughnut)
        mDoughnuts.Add(Value)
    End Set
End Property
```

Visual C#

```
public Doughnut this[int Index]
{
    get
    {
        return mDoughnuts[Index];
    }
    set
    {
        mDoughnuts.Add(value);
    }
}
```

Упражнение 2. Создание дополнительных типов объектов Doughnut

Сейчас вы реализуете дополнительные функции виртуальной машины по производству пончиков и закончите инкапсуляцию функциональности компонента *Doughnut Machine,* скрыв свойство *Enabled* и добавив метод для его установки. Вы также обновите пользовательский интерфейс приложения, чтобы добавить к нему поддержку новых видов пончиков.

▶ Создание дополнительных типов объектов Doughnut

1. Измените уровни доступа свойств *Enabled* и *Intermal* с *Public* (*public*) на *Protected (protected).* Это позволит скрыть эти свойства от внешних классов, инкапсулировав соответствующую функциональность в классе *DoughnutMachine.*
2. После этого свойство *Enabled* станет недоступным для внешних объектов, поэтому придется добавить специальный метод для установки этого свойства:

Visual Basic .NET

```
Public Sub StopDoughnuts()
    Enabled = False
End Sub
```

Visual C#

```
public void StopDoughnuts()
{
    Enabled - false:
}
```

3. Откройте форму *frmMain* в окне дизайнера и добавьте к элементу меню Doughnuts\Make\Cake вложенное меню с элементами Chocolate, Plain и Sugar. Присвойте новым элементам меню имена *mnuCakeChocolate, mnuCakePlain* и *mnuCakeSugar.* Готовое меню показано на рис. 4-1.

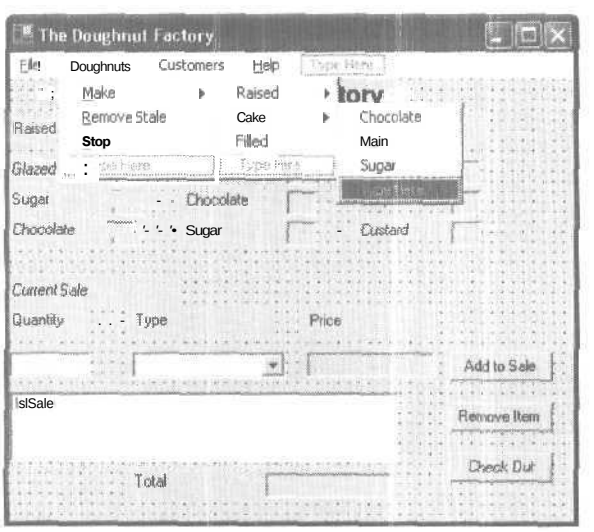


Рис. 4-1. Добавление вложенного меню Cake Doughnuts

4. Добавьте к элементу меню Doughnuts\Make\Filled вложенное меню с элементами Lemon, Grape и Custard. Присвойте новым элементам имена *mnuFilledLemon*, *mnuFilledGrape* и *mnuFilledCustard*. Готовое меню показано на рис. 4-2.

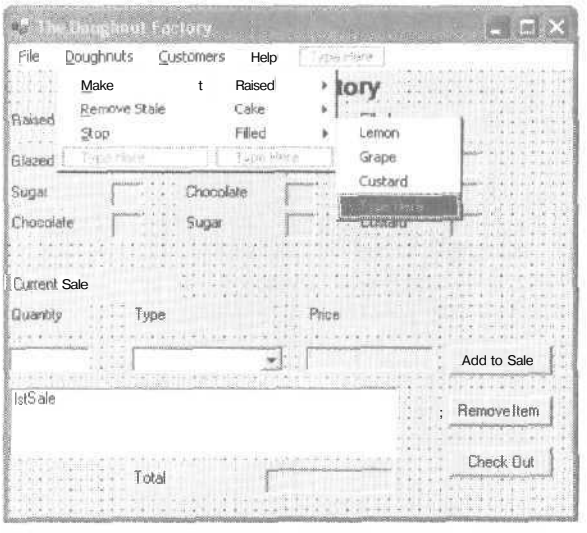


Рис. 4-2. Добавление вложенного меню Filled Doughnuts

5. Найдите код метода *mnuRaisedGlazed_Click*. Замените код, который «вручную» сбрасывает выделенные пункты меню кодом, автоматически перебирает все известные пункты меню и сбрасывает выделение прежде, чем будет выбран другой пункт. Это сделает код для сброса выделения у девяти пунктов меню более управляемым и менее громоздким. Описанный метод реализуется так:

Visual Basic .NET

```
Private Sub mnuRaisedGlazed_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles mnuRaisedGlazed.Click
   Dim item As MenuItem
   ' Перебрать все элементы меню из группы mnuMake
   For Each item In mnuMake.MenuItems
      Dim anotheritem As MenuItem
      ' Снять выделение со всех элементов
      ' меню из группы mnuMake.
      For Each anotheritem In item.MenuItems
         anotheritem.Checked = False
      Next
   Next
   mnuRaisedGlazed.Checked = True
   myDoughnutMachine.MakeDoughnuts(DoughnutMachine.DoughnutType.Glazec)
End Sub
```

Visual C#

```
private void mnuRaisedGlazed_Click(object sender,
   System.EventArgs e)
```

```
{
   // Перебрать все элементы меню из группы mnuMake
   foreach(MenuItem item in mnuMake.MenuItems)
   {
      // Снять выделение со всех элементов
      // меню из группы mnuMake.
      foreach(MenuItem anotheritem in item.MenuItems)
         anotheritem.Checked = false;
   }
   mnuRaisedSugar.Checked = true;
   myDoughnutMachine.MakeDoughnuts(DoughnutMachine.DoughnutType.Glazed);
}
```

6. Аналогично внесите изменения в код методов *mnuRaised Sugar_Click* и *mnuRaised-Chocolate_Click*.
7. Добавьте сходные методы, обрабатывающие события *Click* элементов меню, добавленных в пунктах 3 и 4 этого упражнения,
8. Измените метод *mnuStop_Click* так, чтобы он вызывал метод *Stop Doughnuts* объекта *DoughnutMachine*, например:

Visual Basic .NET

```
Private Sub mnuStop_Click(ByVal sender As System.Object, ByVal e _
   As System.EventArgs) Handles mnuStop.Click
   myDoughnutMachine.StopDoughnuts()
End Sub
```

Visual C#

```
private void mnuStop_Click(object sender, System.EventArgs e)
{
   myDoughnutMachine.StopDoughnuts();
}
```

9. Добавьте к методу *DoughnutCompleteHandler* код, который будет вести учет новых типов пончиков и обновлять соответствующие элементы управления пользовательского интерфейса. Для каждого типа *DoughnutType* должен быть свой оператор *Case,* увеличивающий значение закрытой переменной, которая представляет соответствующий тип *DoughnutType,* и отображающий обновленное значение этой переменной в соответствующем поле формы.
10. Сохраните проект и протестируйте его.

Упражнение 3. Реализация метода RemoveStale

Сейчас вы напишете метод класса *Doughnut Machine,* предназначенный для проверки наличия пончиков, которые «пролежали» в наборе так долго. что успели засохнуть. Этот метод должен удалять «черствые» пончики из набора и обновлять соответствующим образом данные, отображаемые пользовательским интерфейсом. Вам придется добавить к пользовательскому интерфейсу соответствующий код, который будет вызывать этот метод и обновлять форму. А также реализовать структуру, представляющую

группу пончиков различных видов, — она потребуется для возврата сведений об удаленных пончиках.

▶ Создание метода RemoveStale

1. В классе *DoughnutMachine* объявите структуру, которая способна представлять группу любого размера, включающую любые виды пончиков. Вот пример реализации такой структуры:

Visual Basic .NET

```
Public Structure DoughnutGroup
   Public RaisedGlazed As Integer
   Public RaisedSugar As Integer
   Public RaisedChocolate As Integer
   Public CakePlain As Integer
   Public CakeChocolate As Integer
   Public CakeSugar As Integer
   Public FilledLemon As Integer
   Public FilledGrape As Integer
   Public FilledCustard As Integer
End Structure
```

Visual C#

```
public struct DoughnutGroup
{
   public int RaisedGlazed;
   public int RaisedSugar;
   public int RaisedChocolate;
   public int CakePlain;
   public int CakeChocolate;
   public int CakeSugar;
   public int FilledLemon;
   public int FilledGrape;
   public int FilledCustard;
}
```

2. Определите в классе *DoughnutMachine* метод, проверяющий «срок годности» всех пончиков в наборе. Пончики, пролежавшие на складе слишком долго, следует удалить из набора. Этот метод должен возвращать экземпляр структуры *DoughnutGroup*, уведомляющий о количестве и видах удаленных пончиков.

Visual Basic .NET

```
Public Function RemoveStale() As DoughnutGroup
   Dim myGroup As New DoughnutGroup()
   Dim Counter As Integer
   Dim UpperBound As Integer
   UpperBound = mDoughnuts.Count - 1
   Dim myTimeSpan As TimeSpan
   For Counter = 0 To UpperBound
```

```
        myTimeSpan = Now.Subtract(mDoughnuts(UpperBound - _
            Counter).TimeOfCreation)
        If myTimeSpan.TotalMinutes >= 10 Then
            Select Case mDoughnuts(UpperBound - Counter).Flavor
                Case DoughnutType.Chocolate
                    myGroup.RaisedChocolate += 1
                Case DoughnutType.ChocolateCake
                    myGroup.CakeChocolate += 1
                Case DoughnutType.Custard
                    myGroup.FilledCustard += 1
                Case DoughnutType.Glazed
                    myGroup.RaisedGlazed += 1
                Case DoughnutType.Grape
                    myGroup.FilledGrape += 1
                Case DoughnutType.Lemon
                    myGroup.FilledLemon += 1
                Case DoughnutType.PlainCake
                    myGroup.CakePlain += 1
                Case DoughnutType.Sugar
                    myGroup.RaisedSugar += 1
                Case DoughnutType.SugarCake
                    myGroup.CakeSugar += 1
            End Select
            mDoughnuts.RemoveAt(UpperBound - Counter)
        End If
    Next
    Return myGroup
End Function
```

Visual C#

```
public DoughnutGroup RemoveStale()
{
    DoughnutGroup myGroup = new DoughnutGroup();
    int UpperBound;
    UpperBound = mDoughnuts.Count - 1;
    TimeSpan myTimeSpan;
    for(int Counter = 0; Counter <= UpperBound; Counter++)
    {
        myTimeSpan =
            System.DateTime.Now.Subtract(mDoughnuts[UpperBound -
            Counter].TimeOfCreation);
        if(myTimeSpan.TotalMinutes >= 10)
        {
            switch(mDoughnuts[UpperBound - Counter].Flavor)
            {
                case DoughnutType.Chocolate:
                    myGroup.RaisedChocolate ++;
```

```
                break;
            case DoughnutType.ChocolateCake:
                myGroup.CakeChocolate ++;
                break;
            case DoughnutType.Custard:
                myGroup.FilledCustard ++;
                break;
            case DoughnutType.Glazed:
                myGroup.RaisedGlazed ++;
                break;
            case DoughnutType.Grape:
                myGroup.FilledGrape ++;
                break;
            case DoughnutType.Lemon:
                myGroup.FilledLemon ++;
                break;
            case DoughnutType.PlainCake:
                myGroup.CakePlain ++;
                break;
            case DoughnutType.Sugar:
                myGroup.RaisedSugar ++;
                break;
            case DoughnutType.SugarCake:
                myGroup.CakeSugar ++;
                break;
        }
        mDoughnuts.RemoveAt(UpperBound - Counter);
    }
  }
  return myGroup;
}
```

3. Добавьте к форме *frmMain* обработчик события *Click* элемента *mnuRemoveStale*, который при удалении пончиков будет вызывать метод *RemoveStale* объекта *DoughnutMachine* и обновлять соответствующие элементы управления пользовательского интерфейса. Вот пример кода этого метода:

Visual Basic .NET

```
Private Sub mnuRemoveStale_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles mnuRemoveStale.Click
  Dim aDoughnutGroup As DoughnutMachine.DoughnutGroup
  aDoughnutGroup = myDoughnutMachine.RemoveStale()
  mRaisedGlazed = mRaisedGlazed - aDoughnutGroup.RaisedGlazed
  mRaisedSugar - mRaisedSugar - aDoughnutGroup.RaisedSugar
  mRaisedChocolate = mRaisedChocolate - aDoughnutGroup.RaisedChocolate
  mCakePlain = mCakePlain - aDoughnutGroup.CakePlain
  mCakeChocolate = mCakeChocolate - aDoughnutGroup.CakeChocolate
  mCakeSugar = mCakeSugar - aDoughnutGroup.CakeSugar
```

```
    mFilledLemon = mFilledLemon - aDoughnutGroup.FilledLemon
    mFilledGrape = mFilledGrape - aDoughnutGroup.FilledGrape
    mFilledCustard = mFilledCustard - aDoughnutGroup.FilledCustard
    txtGlazedRaised.Text = mRaisedGlazed.ToString
    txtSugarRaised.Text = mRaisedSugar.ToString
    txtchocolateRaised.Text = mRaisedChocolate.ToString
    txtcustardFilled.Text = mFilledCustard.ToString
    txtGrapeFilled.Text = mFilledGrape.ToString
    txtLemonFilled.Text = mFilledLemon.ToString
    txtChocolateCake.Text = mCakeChocolate.ToString
    txtPlainCake.Text = mCakePlain.ToString
    txtSugarCake.Text = mCakeSugar.ToString
End Sub
```

Visual C#

```
private void mnuRemoveStale_Click(object sender, System.EventArgs e)
{
    DoughnutMachine.DoughnutGroup aDoughnutGroup;
    aDoughnutGroup = myDoughnutMachine.RemoveStale();
    mRaisedGlazed = mRaisedGlazed - aDoughnutGroup.RaisedGlazed;
    mRaisedSugar = mRaisedSugar - aDoughnutGroup.RaisedSugar;
    mRaisedChocolate = mRaisedChocolate -
        aDoughnutGroup.RaisedChocolate;
    mCakePlain = mCakePlain - aDoughnutGroup.CakePlain;
    mCakeChocolate = mCakeChocolate - aDoughnutGroup.CakeChocolate;
    mCakeSugar = mCakeSugar - aDoughnutGroup.CakeSugar;
    mFilledLemon = mFilledLemon - aDoughnutGroup.FilledLemon;
    mFilledGrape = mFilledGrape - aDoughnutGroup.FilledGrape;
    mFilledCustard = mFilledCustard - aDoughnutGroup.FilledCustard;
    txtGlazedRaised.Text = mRaisedGlazed.ToString();
    txtSugarRaised.Text = mRaisedSugar.ToString();
    txtchocolateRaised.Text = mRaisedChocolate.ToString0;
    txtcustardFilled.Text = mFilledCustard.ToString();
    txtGrapeFilled.Text = mFilledGrape.ToString();
    txtLemonFilled.Text = mFilledLemon.ToString();
    txtChocolateCake.Text = mCakeChocolate.ToString();
    txtPlainCake.Text = mCakePlain.ToString();
    txtSugarCake.Text = mCakeSugar.ToString();
}
```

4. Сохраните **проект и протестируйте его.**

Закрепление материала

Приведенные ниже вопросы помогут вам лучше усвоить основные темы данной главы. Если вы не сумеете ответить на вопрос, повторите материал соответствующего занятия. Ответы для самопроверки — в приложении «Вопросы и ответы» в конце книги.

1. Дайте краткое определение концепции инкапсуляции и раскройте ее значение для ООП.

2. Что такое перегрузка и для чего она применяется?

3. Вам необходимо создать несколько не связанных между собой классов, предоставляющих общий набор методов. Как, используя принцип полиморфизма, сделать методы, предоставляемые этими классами, доступными другим классам?

4. Вам нужно создать несколько классов, поддерживающих основные возможности, но предназначенных для взаимодействия с разными наборами объектов. Какой способ выбрать, чтобы затратить на разработку таких классов минимальное время?

5. Что такое абстрактные классы и для чего их применяют?

ГЛАВА 5

Тестирование и отладка приложений

Занятие 1. Использование отладочных инструментов 194

Занятие 2. Применение классов Debug и Trace 205

Занятие 3. Создание плана блочного тестирования 215

Занятие 4. Обработка и генерация исключений 220

В этой главе

Из этой главы вы узнаете, как пользоваться инструментами для отладки приложений, применять классы *Trace* и *Debug* для диагностики проблем, возникающих при исполнении программ, создавать эффективные планы блочного тестирования, а также научитесь обрабатывать и генерировать исключения.

Прежде всего

Изучение материалов этой главы не требует специальной подготовки.

Занятие 1. Использование отладочных инструментов

Полностью избежать ошибок при программировании невозможно, даже профессионалы время от времени допускают ошибки. Ошибки в программах часто называют «*жучками*» (bugs), а процесс поиска и исправления этих ошибок — *отладкой* (debugging, буквально — «ловля жучков»). На этом занятии вы познакомитесь с различными типами возможных ошибок и с инструментами для их поиска и исправления.

Изучив материал этого занятия, вы сможете:

- ✓ описать наиболее распространенные типы ошибок;
- ✓ применять режим Break и устанавливать точки прерывания;
- ✓ осуществлять пошаговое исполнение кода;
- ✓ рассказать, как работают с отладочными инструментами среды разработки.

Продолжительность занятия - около 45 минут.

Основные типы ошибок

При написании кода программисты сталкиваются с тремя основными типами ошибок. *Синтаксические ошибки* (syntax errors) возникают, если компилятор не в состоянии «понять» переданный ему исходный текст. При попытке выполнить недопустимое действие во время исполнения программы возникают *ошибки периода выполнения* (run-time errors). Наконец, *логические ошибки* (logical errors) не препятствуют компиляции и исполнению программы, но приводят к неожиданным результатам.

Синтаксические ошибки

Синтаксические ошибки возникают, если компилятор не может скомпилировать предоставленный ему код, например, из-за опечатки в ключевом слове, пропуска знака препинания или использования неверной конструкции. Следующий пример кода содержит две синтаксические ошибки:

Visual Basic .NET

```
Public Sub SyntaxError()
   System.Windows.Forms.MessageBoxShow("Where's the error?")
```

Visual C#

```
public void SyntaxErrorO
{
   System.Windows.Forms.MessageBoxShow("Where's the error?");
```

Во-первых, здесь пропущена точка между MessageBox и Show, поэтому компилятор не распознает такую ссылку. Во-вторых, отсутствует замыкающий элемент конструкции метода (в Visual Basic .NET это *End Sub*, а в Visual C# — *}*). Любая из этих ошибок не позволит компилятору интерпретировать такой код.

Visual Studio .NET облегчает поиск синтаксических ошибок, автоматически обнаруживая их при компоновке проекта и выделяя в исходном тексте подсветкой. Список обнаруженных ошибок также выводится в окне Task List (рис. 5-1).

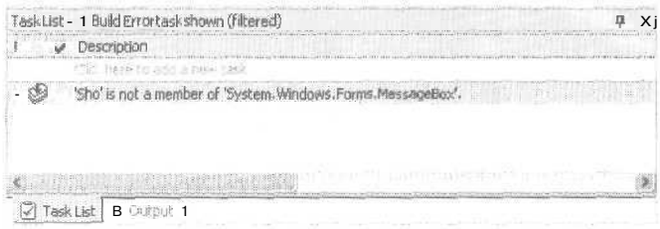


Рис. 5-1. Окно Task List

Если дважды щелкнуть описание ошибки в окне Task List, среда разработки немедленно переместит курсор в соответствующую строку в окне кода и выделит ошибку подсветкой — обычно этого достаточно, чтобы ее распознать и исправить. Чтобы получить дополнительное описание выделенной ошибки, нажмите F1.

Visual Basic .NET автоматически проверяет синтаксис кода по мере его ввода, при этом обнаруженные синтаксические ошибки сразу же выделяются подсветкой и добавляются к списку в окне Task List, таким образом, не обязательно инициировать компоновку проекта, чтобы обнаружить синтаксические ошибки. Visual C# также поддерживает автоматическую проверку синтаксиса кода при вводе, но она не столь совершенна, как в Visual Basic.

▶ **Как найти и исправить допущенные синтаксические ошибки**

1. Выполните компоновку проекта — при этом любые синтаксические ошибки будут обнаружены и добавлены в окно Task List (при использовании Visual Basic .NET это делается автоматически без компоновки проекта).
2. Дважды щелкните ошибку в окне Task List — она будет выделена в окне редактора кода подсветкой.
3. Исправьте найденную ошибку. Если требуется дополнительная информация, нажмите F1.

Ошибки периода выполнения

Они возникают при попытке приложения выполнить недопустимую операцию. К этой категории относятся деление на ноль и действия, запрещенные политиками безопасности: попытка их исполнения приводит к генерации исключения защиты. При возникновении ошибки периода выполнения генерируется *исключение* (exception), предоставляющее описание ошибки, — специальный класс, который служит для передачи сведений об ошибке другим компонентам приложения. Можно написать код, который будет обрабатывать исключения, не допуская остановки исполнения приложения. Подробнее обработка исключений рассматривается на занятии 4.

Логические ошибки

Логическую ошибку констатируют, если приложение корректно компилируется и исполняется, но не выдает ожидаемые результаты. Диагностировать такие ошибки труднее, чем остальные, из-за отсутствия указаний на источник ошибки. Причиной логической ошибки могут быть такие, на первый взгляд, незначительные промахи, как неверно поставленная точка в десятичной дроби или лишний ноль:

Visual Basic .NET

```
Public Function CalculateSalary(ByVal HourlyRate As Single) As Single
   Dim YearlySalary As Single
   YearlySalary = HourlyRate * 52 * 400
   Return YearlySalary
End Function
```

Visual C#

```
public float CalculateSalary(float HourlyRate)
{
   float YearlySalary;
   YearlySalary = HourlyRate * 52 * 400;
   return YearlySalary;
}
```

Этот метод получает значение ставки почасовой оплаты и вычисляет размер годовой зарплаты, умножая полученное значение на число недель в году и на количество рабочих часов в неделю. Ошибка возникает, поскольку значение почасовой оплаты умножается на 400, тогда как в рабочей неделе 40 часов. Компиляция и исполнение этого метода проходит безошибочно, но результат он возвращает неверный. Чтобы обнаружить логическую ошибку, необходимо предоставить приложению тестовые данные и проанализировать полученные результаты. Подробнее об этом рассказывается на занятии 3. Часто для «отлова» логической ошибки приходится анализировать код, строку за строкой. Для этого применяется режим пошагового исполнения кода.

Пошаговое исполнение кода

В этом режиме программу можно останавливать после исполнения каждой строки кода, а также просматривать значения переменных и свойств приложения. Visual Studio .NET переходит в режим пошагового исполнения в следующих обстоятельствах:

* при выборе в меню команды Step Into, Step Over или Step Out, а также при нажатии соответствующей кнопки на инструментальной панели Debug;
* по достижении строки с активной точкой прерывания при исполнении программы;
* при исполнении оператора *Stop:*
* при возникновении необработанного исключения.

В режиме пошагового исполнения код просматривают при помощи различных отладочных инструментов. В таблице 5-1 перечислены отладочные функции, доступные через меню Debug.

Таблица 5-1. Элементы меню Debug

Элемент меню	Быстрая клавиша	Описание
Windows		Открывает доступ к командам, вызывающим различные отладочные инструменты. Более подробно эти инструменты описаны далее

Таблица 5-1. ***(окончание)***

Элемент меню	Быстрая клавиша	Описание
Start/Continue	F5	Запускает приложение в режиме отладки; если приложение находится в режиме пошагового исполнения, продолжает исполнение программы
Break All	Ctrl+Alt+Break	Останавливает исполнение программы на текущей строке и переводит ее в режим пошагового исполнения. Чтобы возобновить исполнение, выберите команду Continue
Stop Debugging	Shift+F5	Останавливает отладчик и возвращает Visual Studio в режим Design
Detach All		Отключает отладчик от всех процессов и закрывает его, не останавливая исполнения программы
Restart	Ctrl+Shift+F5	Перезапускает приложение
Apply Code Changes		Применяется только с программами на C/C++, недоступен для программ на Visual Basic .NET и Visual C#
Processes		Открывает окно Processes
Exceptions	Ctrl+Alt+E	Открывает окно Exceptions
Step Into	F8	Исполняет следующую строку; если она содержит вызов метода, останавливает исполнение в начале этого метода
Step Over	Shift+F8	Исполняет следующую строку; если она содержит вызов метода, исполняет его и останавливает исполнение на следующей строке текущего метода
Step Out	Shift+F11	Исполняет код до конца текущего метода и останавливается перед следующей исполняемой строкой вызывающего метода
QuickWatch	Ctrl+Alt+Q	Открывает окно QuickWatch
New Breakpoint	Ctrl+B	Открывает окно Breakpoint
Clear All Breakpoints	Ctrl+Shift+F9	Удаляет из приложения все точки прерывания
Disable All Breakpoints		Деактивирует все точки прерывания, не удаляя их

Ряд отладочных средств (таблица 5-2) доступен через контекстное меню окна кода,

Таблица 5-2. Отладочные средства, доступные через контекстное меню окна кода

Элемент меню	Описание
Insert Breakpoint	Вставляет точку прерывания в выделенную строку
New Breakpoint	Открывает окно New Breakpoint; функциональный эквивалент одноименной команды из меню Debug

Таблица 5-2. ***(окончание)***

Элемент меню	Описание
Add Watch	Добавляет выделенное выражение в окно Watch
QuickWatch	Открывает окно QuickWatch, функциональный эквивалент одноименной команды из меню Debug
Show Next Statement	Выделяет подсветкой оператор, который будет исполнен следующим
Run To Cursor	Исполняет программу до выделенной строки
Set Next Statement	Назначает выделенную строку для исполнения (выделенная строка должна быть в текущей процедуре)

Наиболее востребованные отладочные средства описаны в следующих разделах.

Step Into

Команда Step Into (исполнение с заходом) позволяет анализировать работу программы, исполняя ее построчно. При ее выборе исполняется текущая строка кода и исполнение останавливается. Если следующая строка содержит вызов метода, отладчик исполняет первую строку этого метода, после чего останавливает программу, позволяя проанализировать состояние приложения с помощью отладочных инструментов.

Чтобы воспользоваться командой Step Into, выберите ее из меню Debug, либо щелкните соответствующую кнопку на инструментальной панели Debug, либо нажмите F11.

Step Over

Команда Step Over (исполнение без захода) исполняет следующую строку кода, как и Step Into, но если в этой строке будет вызов метода, команда Step Over исполняет его до конца и останавливает исполнение перед следующей строкой текущей процедуры. Step Over позволяет оставаться на одном и том же уровне кода, не заходя внутрь вложенных методов. Команда удобна, когда не требуется анализировать вложенные методы.

Чтобы воспользоваться Step Over, выберите ее из меню Debug, либо щелкните соответствующую кнопку на инструментальной панели Debug, либо нажмите F10.

Step Out

Команда Step Out (исполнить текущий метод) позволяет исполнить текущую процедуру до конца. Если она вызвана из другого метода, исполнение останавливается перед оператором, следующим за строкой, откуда была вызвана текущая процедура.

Чтобы воспользоваться командой Step Out, выберите ее из меню Debug, либо щелкните соответствующую кнопку на инструментальной панели Debug, либо нажмите Shift+F11.

Run To Cursor

Команда Run To Cursor позволяет выбрать строку кода и выполнить весь код приложения до этой строки. Эта команда позволяет пропустить блоки кода, которые не нужно исполнять построчно, например длинный цикл.

▶ **Как воспользоваться командой Run To Cursor**

В редакторе исходного текста поместите курсор на строку, до которой следует продолжить исполнение. Эта строка должна быть строкой исполняемого кода. Щелкните правой кнопкой мыши и выберите команду Run To Cursor.

Set Next Statement

Команда Set Next Statement позволяет пропускать отдельные участки кода. Вы сможете назначить оператор, который будет исполнен следующим, при этом пропускается весь код, расположенный до этого оператора, — этим команда Set Next Statement отличается от Run To Cursor. Используя Set Next Statement, удается пропустить заведомо ошибочные участки кода, а также повторно исполнять ранее исполненные строки кода.

Чтобы воспользоваться командой Set Next Statement, перейдите в окно редактора кода и поместите курсор в строку, которую следует исполнить следующей (она должна быть внутри текущей процедуры). Затем щелкните эту строку правой кнопкой мыши и выберите из контекстного меню команду Set Next Statement, Чтобы исполнить помеченную таким образом строку, воспользуйтесь командой Step Into.

Установка точек прерывания

Можно назначать определенные строки кода или задавать условия, при исполнении которых отладчик непременно остановит исполнение приложения. Такие процедуры называются установкой *точек прерывания* (breakpoints), они позволяют останавливать исполнение программы в заданном месте или при определенных обстоятельствах. Точки прерывания задаются:

- **по функции.** Они останавливают исполнение по достижении определенной строки функции;
- **по адресу в файле.** Останавливают исполнение приложения по достижении определенного места в файле исходного текста;
- **по адресу памяти.** Останавливают приложение при обращении к определенному адресу памяти;
- **по данным.** Останавливают приложение при изменении значения заданной переменной; недоступны при использовании Visual Basic .NET и Visual C#.

Точки прерывания первого типа применяются чаще всего, их устанавливают одним из трех способов:

- щелкнув серую область в левой части окна кода напротив нужной строки — в результате точка прерывания устанавливается в этой строке ;
- щелкнув правой кнопкой нужную строку и выбрав из контекстного меню команду Insert Breakpoint;
- выбрав команду New Breakpoint из меню Debug или из контекстного меню редактора кода и установив соответствующие параметры точки прерывания в окне New Breakpoint.

При установке новой точки прерывания через окно New Breakpoint можно задать условия, определяющие, будет ли остановлено исполнение приложения по достижении этой точки прерывания. Если щелкнуть кнопку Condition, откроется окно Breakpoint Condition, где предлагается ввести некоторое выражение. В результате точка прерывания активизируется, если при вычислении выражения получится *true* либо при изменении выражения. Кнопка Hit Count открывает окно Break-

point Hit Count, где задают значение счетчика, по достижении которого точка прерывания активизируется,

Окно Breakpoints

Окно Breakpoints (рис. 5-2) обеспечивает централизованное управление всеми точками прерывания.

Здесь отображаются все точки прерывания, установленные в проекте с указанием их размещения и условий, определенных для каждой точки прерывания. В меню Columns выбирают для отображения дополнительные столбцы со сведениями о точках прерывания. В этом окне также можно деактивировать точку прерывания, сняв соответствующий флажок. Кроме того, кнопка в верхнем левом углу этого окна позволяет создать новые и удалить существующие точки прерывания, а также очистить список или отключить все точки прерывания.

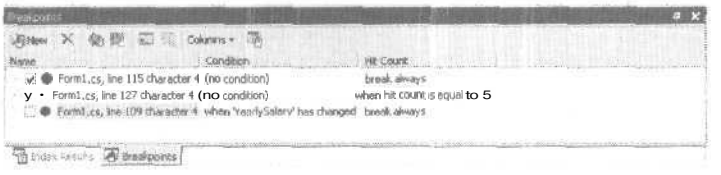


Рис. 5-2. Окно Breakpoints

Удаление и отключение точек прерывания

Точку прерывания, ставшую ненужной, можно удалить либо отключить, если она потребуется в дальнейшем. Как уже говорилось, для управления точками прерывания предназначено окно Breakpoints. Вы также можете удалить или перевести в неактивное состояние точку прерывания в редакторе кода.

Чтобы удалить ненужную точку прерывания, щелкните ее на серой панели слева от редактора кода.

Чтобы отключить точку прерывания, щелкните ее правой кнопкой мыши в редакторе кода и выберите Disable Breakpoint. Таким образом, она будет переведена в неактивное состояние.

Применение отладочных инструментов

Visual Studio предоставляет несколько инструментов, позволяющих отслеживать исполнение программ. Одни из них, такие, как окна Output и Command, доступны в режиме Design, а другие, например окно Locals, видно только при отладке. В этом разделе описаны некоторые окна, наиболее часто используемые при отладке.

Окно Output

В окне Output отображаются все данные, которые приложение выводит в командной строке в ходе его компиляции и исполнения: операторы *Debug* и *Trace,* а также сообщения, которыми приложение уведомляет о загрузке сборок. При отладке приложения в окне Output, как правило, просматривают уведомления оператора *Debug,* о котором более подробно рассказывается на занятии 2. По умолчанию окно Output видно; в противном случае его можно вызвать, выбрав из меню команду View\Other Windows\Output.

Окна Locals, Autos и Watch

Окна Locals, Autos и Watch позволяют отслеживать и редактировать значения переменных программы во время ее отладки, что удобно при тестировании процедур путем передачи им различных аргументов.

Окно Locals

При исполнении программы в режиме отладки (Debug) в окне Locals можно отслеживать значения всех переменных текущей процедуры. В нем отображаются три столбца: Name (с именем переменной), Value (с ее значением) и Type (тип переменной). Сложные переменные, например экземпляры классов и структур, показаны в виде дерева с узлами, раскрывая которые, вы получаете доступ к значениям их членов, При передаче управления от метода к методу содержимое окна Locals меняется, отображая только локальные переменные. В окне Locals Вы можете изменять значение переменной. Для этого следует выделить в столбце Value значение нужной переменной и ввести новое значение. Заметьте, что так разрешено корректировать только значения строковых и числовых переменных. Модифицировать переменные-объекты, помещая в них ссылки на другие экземпляры класса или структуры, нельзя, зато можно изменять значения их членов. Измененные значения в столбце Value подсвечиваются красным цветом.

Чтобы отобразить окно Locals во время отладки приложения, выберите из меню Debug команду Windows\Locals.

Окно Autos

Окно Autos напоминает окно Locals, но его возможности ограничены. В нем отображаются те же столбцы, что и в окне Locals, но в них содержатся только переменные из текущей и предыдущей строк кода (в Visual C#) либо (в Visual Basic .NET) переменные из текущей строки плюс три строки до и после нее. В окне Autos можно изменять значение переменных так же, как и в Locals.

Чтобы вызывать окно Autos во время отладки приложения, выберите из меню Debug команду Windows\Autos.

Окно Watch

В окно Watch можно добавить переменную, чтобы отслеживать, как изменяются ее значения. Переменные не удаляются из окна Watch, даже когда они уходят из области видимости. Как и в Locals и Autos, в окне Watch отображаются столбцы Name, Value и Type, содержащие сведения о переменных; здесь также разрешается изменять текущие значения простых переменных путем ввода новых значений в столбец Value. Visual Studio .NET позволяет открывать до четырех окон Watch, что дает возможность отслеживать одновременно несколько наборов переменных,

Диалоговое окно QuickWatch предусмотрено для быстрого расчета значения переменной. В столбцах Name, Value и Type этого окна отображаются сведения только об одной переменной; из него переменную можно добавить в окно Watch, а также изменить ее значение, введя новое значение в столбец Value.

- Чтобы вызвать окно Watch во время отладки приложения, выберите из меню Debug команду Windows, затем одну из следующих команд: Watch1, Watch2, Watch3 или Watch4.
- Чтобы вызвать диалоговое окно QuickWatch, щелкните правой кнопкой переменную в окне редакторе кода и выберите из контекстного меню команду QuickWatch.

- Добавить переменную в окно Watch можно одним из следующих способов:
  - щелкните пустую строку в окне Watch и введите в нее имя переменной;
  - в редакторе кода щелкните правой кнопкой переменную и из контекстного меню выберите команду Add Watch;
  - щелкните переменную в редакторе кода, окнах Autos либо Locals и перетащите ее в окно Watch;
  - в диалоговом окне QuickWatch щелкните кнопку Add Watch.

Окно Command в режиме Immediate

Этот режим окна Command применяют при отладке для исполнения процедур, расчета значений выражений или изменения значений переменных.

Окно Immediate позволяет исполнить любой допустимый оператор, но не разрешает объявлять переменные и объекты. В режиме пошагового исполнения кода можно ввести в окно Command оператор или имя метода так же, как в редакторе кода. Нажав Enter, вы заставите Visual Studio исполнить введенный оператор. Завершив исполнение оператора или метода, приложение вернется в режим пошагового исполнения.

Когда приложение находится в режиме пошагового исполнения, значения выражений и переменных можно вводить в окно Command напрямую, предваряя их знаком вопроса. Например, таким образом перемножить текущие значения X и Y, получив результат в окне Command:

```
? X*Y
```

Обратите внимание на использование одинакового синтаксиса для Visual Basic .NET и Visual C# в окне Command. Скажем, в C# строку не обязательно завершать точкой с запятой. Унифицированный синтаксис позволяет получить значение любого свойства, которое в настоящей момент находится в области видимости. Например, следующая команда

```
? TextBox1.Text
```

выводит текущее значение свойства *Text* объекта *TextBox1*.

Оператор присваивания позволяет присвоить существующей переменной новое значение. Так, следующий пример присваивает переменной *X* значение 5:

```
X = 5
```

Заметьте: эту переменную необходимо объявить заранее.

Чтобы вызвать окно Command, укажите в меню View команду Other Windows\Command Window. Чтобы открыть окно Command в режиме Immediate, щелкните команду Debug\Windows\Immediate.

Другие отладочные окна

Дополнительные окна отображают сведения об исполнении программы во время отладки и позволяют управлять пошаговым исполнением кода. Они доступны через меню Debug (таблица 5-3).

Таблица 5-3. Дополнительные окна, доступные через меню Debug

Название	Описание
Running Documents	Отображает список текущих документов, загруженных в исполняемый процесс

Таблица 5-3. *(окончание)*

Название	Описание
Me (Visual Basic .NET) / This (Visual C#)	Отображает члены данных проекта, связанные с текущим методом
Call Stack	Отображает имена функций из стека вызовов с указанием типов и значений их параметров
Threads	Позволяет просматривать потоки отлаживаемой программы и управлять ими
Modules	Отображает список используемых программой модулей (DLL- и EXE-файлов)
Memory	Отображает содержимое памяти. Применяется для просмотра больших буферов, строк и других данных, которые нельзя отобразить в окнах Locals, Autos или Watch
Disassembly	Показывает код сборки, соответствующий сгенерированным компилятором командам
Registers	Отображает содержимое регистров. Оставив это окно открытым при пошаговом исполнении программы, вы сможете наблюдать за изменением значений регистров во время исполнения

Резюме

- При написании программ возможны три типа ошибок:
  - синтаксические ошибки возникают, если компилятор не в состоянии интерпретировать синтаксис предоставленного ему кода;
  - ошибки периода выполнения возникают при попытке выполнить недопустимое действие;
  - логические ошибки не препятствуют компиляции и исполнению кода, но заставляют его возвращать неожиданные результаты.
- Режим пошагового исполнения кода позволяет построчно анализировать программу для диагностики и исправления ошибок. Visual Studio .NET предоставляет несколько вариантов пошагового исполнения:
  - Step Into позволяет построчно просматривать код с заходом в вызываемые функции;
  - Step Over позволяет построчно просматривать код без захода в вызываемые функции;
  - Step Out исполняет текущую функцию до конца и останавливается (если возможно) на следующей строке функции, из которой была вызвана текущая процедура;
  - Run To Cursor позволяет установить курсор в некоторую строку и исполнить весь код до этой строки;
  - Set Next Statement (назначить следующий оператор) позволяет назначить следующий оператор для исполнения, при этом все строки до этого оператора будут пропущены.

- Точки прерывания — это строки кода, назначенные во время отладки, по достижении которых исполнение останавливается, а приложение переходит в режим пошагового исполнения. Для точек прерывания можно назначать дополнительные условия, определяющие обстоятельства, при которых эти точки активируются. Для управления точками прерывания предназначено окно Breakpoints, позволяющее создавать, отключать и удалять их.
- Visual Studio .NET предоставляет ряд инструментов для наблюдения за исполнением программы. Окна Locals, Autos и Watch позволяют отслеживать значения переменных программы во время ее исполнения; окно Command позволяет исполнять код и получать значения переменных и свойств. Имеются также дополнительные окна для наблюдения за самыми разными данными приложения.

Занятие 2. Применение классов Debug и Trace

Классы *Debug* и *Trace* позволяют генерировать и записывать в журналы сообщения со сведениями о состоянии приложения, не прерывая его исполнения. На этом занятии вы научитесь применять классы *Debug и Trace,* добавлять слушатели трассировки (Trace Listeners) и переключатели, а также настраивать переключатели и компилированной программе.

Изучив материал этого занятия, вы сможете:

- ✓ отображать сообщения об ошибках с помощью классов *Debug и Trace*;
- ✓ создавать слушатели трассировки и записывать трассировочные данные в журнал;
- ✓ создавать и применять трассировочные переключатели;
- ✓ настраивать трассировочные переключатели через конфигурационный файл приложения.

Продолжительность занятия - около 45 минут.

Обычно для отладки несложных приложений хватает возможностей, описанных на предыдущем занятии, например пошагового исполнения кода, но для отладки более сложных приложений этого мало. Большие программы состоят из сотен тысяч строк кода, если такая программа дает неправильные результаты, найти логическую ошибку путем построчного просмотра кода и проверки значений переменных приложения не удастся. Класс *Debug* позволяет генерировать и записывать в журнал во время выполнения сообщения с описанием состояния программы, а класс *Trace* — создавать инструменты для диагностики даже скомпилированных приложений.

Как работает трассировка

Классы *Trace* и *Debug* содержат общие (статические) методы, которые позволяют проверять условия во время выполнения и записывать результаты в журнал. Данные, генерируемые ими, отображаются в окне Output, где их можно просматривать при отладке, и записываются в набор *Listeners.* Набор *Listeners* содержит группу классов, получающих данные от *Trace* и *Debug*; их называют *слушателями.* Классы *Trace* и *Debug* используют общий набор *Listeners,* предоставляя одни и те же данные всем слушателям из этого набора. Разные виды слушателей записывают данные и текстовые файлы или журналы событий. После завершения исполнения программы трассировочные данные, записанные слушателями, анализируют, чтобы найти ошибки в приложении. Трассировка полезна и при настройке программ (об этом рассказывается в главе 9).

В сущности, классы *Trace* и *Debug* идентичны: они предоставляют одинаковые наборы методов и свойств и записывают результаты в набор *Listeners.* Единственное различие между ними в том, что по умолчанию операторы *Debug* удаляются при компиляции окончательной версии сборки, а операторы *Trace* там остаются По-этому класс *Debug* обычно применяют для отладки на стадии разработки, а *Trace* — для тестирования и настройки после компиляции окончательной версии приложения.

Запись данных, генерируемых классами Trace и Debug

Для записи генерируемых классами *Trace* и *Debug* данных в набор *Listeners* служат методы этих классов:

- ***Write*. Записывает строку в набор *Listeners*;**
- ***WriteLine*. Записывает в набор *Listeners* строку, завершая ее символом возврата каретки;**
- ***WriteIf*. Записывает строку в набор *Listeners*, если заданное булево выражение имеет значение *true*;**
- ***WriteLineIf*. Записывает в набор *Listeners* строку, завершенную символом возврата каретки, если заданное булево выражение имеет значение *true*;**
- ***Assert*. Записывает сообщение в набор *Listeners*, если заданное булево выражение имеет значение *false*, и открывает окно с текстом этого сообщения;**
- ***Fail*. Создает проверку, которая автоматически завершается неудачей без проверки условия. При этом в набор *Listeners* записывается сообщение и отображается окно с текстом этого сообщения.**

Следующий пример демонстрирует вызов этих методов в программе:

Visual Basic .NET

```
' Все методы и свойства классов Trace и Debug являются статическими,
' поэтому создавать экземпляры этих классов не требуется.
' Записать строку в набор Listeners,
Trace.Write("Trace Message 1")
' Записать в набор Listeners строку, завершающуюся символом возврата каретки.
Trace.WriteLine("Trace Message 2")
' Записать строку, если заданное выражение имеет значение true.
Debug.WriteIf(X=Y, "X equals Y")
' Записать строку, завершающуюся символом возврата каретки,
' если заданное выражение имеет значение true,
Debug.WriteLineIf(X=Y, "X equals Y")
' Записать результат и показать сообщение,
' если проверка заданного условия дала значение false.
Trace.Assert(X=Y, "X does not equal Y!")
' Безусловно записать результат и показать сообщение-
Debug.Fail("Drive B is no longer valid.")
```

Visual C#

```
// Все методы и свойства классов Trace и Debug являются статическими.
// поэтому создавать экземпляры этих классов не требуется.
// Записать строку в набор Listeners.
Trace.Write("Trace Message 1");
// Записать в набор Listeners строку, завершающуюся символом
// возврата каретки.
Trace.WriteLine("Trace Message 2");
// Записать строку, если заданное выражение имеет значение true.
Debug.WriteIf(X==Y, "X equals Y");
// Записать строку, завершающуюся символом возврата каретки,
```

```
// Если заданное выражение имеет значение true.
Debug.WriteLineIf(X==Y, "X equals Y");
// Записать результат и показать сообщение,
// если проверка заданного условия дала значение false.
Trace.Assert(X==Y, "X does not equal Y!");
// Безусловно записать результат и показать сообщение.
Debug.Fail("Drive 8 is no longer valid.");
```

Можно изменять отступы трассировочных сообщений, записываемых в набор *Listeners,* вызывая методы *Indent* и *Unindent* и устанавливая свойства *IndentSize* и *IndentLevel.* Эти методы и свойства позволяют создать иерархию сообщений об ошибках. Свойство *IndentSize* возвращает или устанавливает число пробелов, составляющее один уровень отступа, а *IndentLevel* — уровень отступа для текущего сообщения. Метод *Indent* увеличивает, а *Unindent* —уменьшает значение *IndentLevel* на единицу.

Чтобы записать результат, используя класс *Trace* или *Debug,* выполните следующие действия.

1. Установите нужный отступ с помощью методов *Indent* и *Unindent,* например, так:

Visual Basic .NET

```
' Увеличить значение свойства IndentLevel на единицу,
Trace.Indent()
```

Visual C#

```
// Увеличить значение свойства IndentLevel на единицу.
Trace.Indent();
```

2. Чтобы записать данные в набор *Listeners,* вызовите соответствующий метод:
   - *Write или WriteLine* для безусловной записи;
   - *WriteIf* или *WriteLineIf,* чтобы записать данные, если проверка условия дает *true;*
   - *Fail* для безусловной записи результата и отображения сообщения;
   - *Assert,* чтобы записать данные, если проверка условия дает *true,* и отобразить сообщение.

Набор Listeners

Все данные, сгенерированные классами *Trace* и *Debug,* направляются в набор *Listeners.* Это специальный набор, упорядочивающий классы, способные получать данные от *Trace,* и предоставляющий доступ к этим классам. Каждый член набора *Listeners* в полном объеме получает данные, генерированные классами *Trace* и *Debug,* способ обработки этих данных зависит от слушателя.

Набор *Listeners* инициализируется единственным членом, экземпляром класса *DefaultTraceListener.* Как видно из имени этого класса, он автоматически создается и получает данные от *Trace* и *Debug* даже в отсутствие других слушателей, добавленных явно. Полученные данные *DefaultTraceListener* передает отладчику и отображает в окне Output среды разработки Visual Studio. Чтобы создать журнал трассировочных сообщений, с которым можно работать независимо от отладчика, необходимо добавить по крайней мере один объект *Listener.* В библиотеке базовых классов

.NET Framework есть два класса, позволяющих создать журнал *Trace — EventLogTraceListener* и *TextWriterTraceListener*. Оба они являются потомками абстрактного .NET-класса *Trace Listener*.

Запись трассировочных данных в текстовом формате

Данные, генерируемые классом *TextWriterTraceListener*, записываются в текстовый файл, в объект *Stream* либо в объект *TextWriter*. Например, у класса *TextWriter* есть свойство *Console.Out*, позволяющее создать класс *TextWriterTraceListener*, который записывает данные в окно Console. Объекты *Stream* и *TextWriter* также применяются для записи данных в текстовые файлы. Чтобы создать объект *TextWriterTraceListener* для записи данных, генерированных классом *Trace*, в текстовый файл, прежде необходимо создать или открыть файл, куда будут записываться данные. После этого следует создать экземпляр *TextWriterTraceListener*, передав ему созданный или открытый файл. В завершение следует добавить *TextWriterTraceListener* к набору *Listeners*. Вот как это делается:

Visual Basic .NET

```
' Открыть заданный файл или создать его, если он не существует,
Dim myLog As New System.IO.FileStream("C:\myFile.txt", _
   IO.FileMode.OpenOrCreate)
' Создать новый объект слушателя и приказать ему
' записывать выходные данные в файл myLog.
Dim myListener As New TextWriterTraceListener(myLog)
' Добавить объект myListener к набору Listeners.
Trace.Listeners.Add(myListener)
```

Visual C#

```
// Открыть заданный файл или создать его, если он не существует.
// Заметьте: путь к файлу необходимо указывать при помощи двойного слэша,
// поскольку слэш является управляющим символом в C#.
System.IO.FileStream myLog = new System.IO.FileStream("C:\\myFile.txt",
   System.IO.FileMode.OpenOrCreate);
// Создать новый объект слушателя и приказать ему
// записывать выходные данные в файл myLog.
TextWriterTraceListener myListener = new
   TextWriterTraceListener(myLog);
// Добавить объект inyListener к набору Listeners.
Trace.Listeners.Add(myListener);
```

Примечание Если открыть существующий файл с помощью метода *IO.FileMode.OpenOrCreate*, содержимое файла будет перезаписано. Чтобы добавить новые данные к существующему файлу, объявляйте объект *FileStream* с помощью метода *IO.FileMode.Append*.

При исполнении кода из предыдущего примера все данные классов *Trace* и *Debug* будут записаны в объект *myListener*. Чтобы на самом деле записать эти данные в файл, необходимо сбросить буфер объекта *Trace*, вызвав метод *Flush* следующим образом:

Visual Basic .NET

```
Trace.Flush()
```

Visual C#

```
Trace.Flush();
```

Можно также установить свойство *TraceAutoFlush* в *true* (см. пример далее), это заставит сбрасывать буфер на диск после каждой записи в него.

Visual Basic .NET

```
Trace.AutoFlush - True
```

Visual C#

```
Trace.AutoFlush = true;
```

Чтобы записать выходные данные объектов *Trace* и *Debug* в текстовый файл, сделайте следующее.

1. Создайте экземпляр объекта *FileStream*, указывающий на нужный текстовый файл.
2. Создайте экземпляр *TextWriterTraceListener* передав ему новый объект *FileStream*.
3. Добавьте созданный объект *Listener* к набору *Listeners*.
4. Установите свойство *Trace.AutoFlush* в *true* или вызовите метод *Trace.Flush* после записи данных.

Запись трассировочных данных в журнал событий

Для записи трассировочных данных в объект *EventLog* применяют объект *EventLogTraceListener*. В сущности, эта методика не отличается от записи в текстовый файл, Необходимо создать объект *EventLogTraceListener*, передав ему новый объект *EventLog* (его следует создать заранее) или ссылку на существующий объект журнала событий, и добавить его к набору *Listeners*. После этого результаты *Trace* будут записываться в указанный объект *EventLog* в виде объектов *EventLogEntry*.

▶ **Запись трассировочных данных в объект *EventLog***

1. Создайте экземпляр объекта *EventLog* и передайте ему новый или существующий.

Visual Basic .NET

```
' Создайте новый журнал событий; "Debug Log" - это отображаемое
' имя журнала.
Dim myLog As New EventLog("Debug Log")
```

Visual C#

```
// Создайте новый журнал событий; "Debug Log" - это отображаемое
// имя журнала.

// EventLog myLog = new EventLog("Debug Log");
```

2. Во избежание ошибки установите свойство *Source* объекта *EventLog*.

Visual Basic .NET

```
myLog.Source = "Trace Output"
```

Visual C#

```
myLog.Source - "Trace Output":
```

3. Создайте новый экземпляр класса *EventLogTraceWriter*, передав ему новый журнал событий для записи трассировочных данных.

Visual Basic .NET

```
Dim myListener As New EventLogTraceListener(myLog)
```

Visual C#

```
EventLogTraceListener myListener = new EventLogTraceListener(myLog);
```

4. При необходимости установите свойство *TraceAutoFlush* в *true* или вызывайте метод *Trace.Flush* после каждой записи.

Применение трассировочных переключателей

Данные, генерируемые операторами *Debug* и *Trace*, интенсивно используются при отладке, но после компиляции и выпуска окончательной версии приложения трассировку включают только в особых случаях. Включать и выключать отображение данных, генерируемых операторами *Trace*, можно средствами трассировочных переключателей. Трассировочными переключателями компилированного приложения управляют через его конфигурационный файл.

Библиотека базовых классов .NET Framework содержит два типа трассировочных переключателей. Первый, *BooleanSwitch*, возвращает значение типа *Boolean*, что делает его похожим на обычный переключатель, не имеющий промежуточных положений. *TraceSwitch* поддерживает параметр с пятью возможными значениями, позволяющий установить для переключателя нужный тип результата.

Экземпляры классов *BooleanSwitch* и *TraceSwitch* создаются так же, как экземпляры любого другого класса. Конструкторы обоих типов переключателей требуют два параметра: *DisplayName*, определяющий имя переключателя, отображаемое в пользовательском интерфейсе, и *Description*, который задает краткое описание переключателя, например:

Visual Basic .NET

```
Dim myBooleanSwitch As New BooleanSwitch("Switch1", _
    "Controls Data Tracing")
Dim nyTraceSwitch As New TraceSwitch("Switch2", _
    "Controls Forml Tracing")
```

Visual C#

```
BooleanSwitch myBooleanSwitch = new BooleanSwitch("Switch1",
    "Controls Data Tracing");
TraceSwitch myTraceSwitch = new TraceSwitch("Switch2",
    "Controls Forml Tracing");
```

Класс *TraceSwitch* поддерживает свойство *TraceSwitch.Level*, значения которого представляют различные уровни ошибок. Это свойство способно принимать любое из пяти значений перечисления *TraceLevel*:

- *TraceLevel. Off* (целочисленное значение 0) деактивирует объект *TraceSwitch*;
- *Trace Level. Error* (целочисленное значение 3) определяет вывод кратких сообщений об ошибках;
- *TraceLevel. Warning* (целочисленное значение 2) позволяет выводить сообщения об ошибках и предупреждения;
- *TraceLevel. Info* (целочисленное значение 3) разрешает выводить сообщения об ошибках, предупреждения и краткие информационные сообщения;
- *TraceLevel. Verbose* (целочисленное значение 5) позволяет выводить сообщения об ошибках, предупреждения и подробные сведения об исполнении программы.

У класса *TraceSwitch* также есть четыре доступных только для чтения свойства типа *Boolean*, представляющих четыре одноименных уровня трассировки:

- *TraceSwitch. TraceError*
- *TraceSwitch. TraceWarning*
- *TraceSwitch. TraceInfo*
- *TraceSwitch. Trace Verbose*

При установке свойства *TraceSwitch.Level* четыре перечисленных свойства автоматически получают соответствующие значения. Например, если *TraceSwitch.Level= Info*, свойства *TraceSwitch.TraceInfo*, *TraceSwitch. TraceError* и *TraceSwitch, TraceWarning* возвращают *true*. После определения значения *Level* свойства, соответствующие данному и более низким уровням возврата ошибок, будут возвращать *true*.

С помощью трассировочных переключателей операторы *Trace* проверяют, следует ли записывать трассировочные данные. Методы *Trace.WriteIf* и *Trace. WriteLine If* позволяют проверить, активен ли переключатель трассировки, и узнать определяемый им уровень трассировки:

Visual Basic .NET

```
' Этот пример предполагает наличие объектов BooleanSwitch и TraceSwitch
' с именами myBooleanSwitch и myTraceSwitch соответственно.
Trace.WriteIf(myBooleanSwitch.Enabled = True, "Error.")
Trace.WriteLineIf(myTraceSwitch.TraceInfo = True, _
   "Type Mismatch Error")
```

Visual C#

```
// Этот пример предполагает наличие объектов BooleanSwitch и TraceSwitch
// с именами myBooleanSwitch и myTraceSwitch соответственно.
Trace.WriteIf(myBooleanSwitch.Enabled == true, "Error");
Trace.WriteLineIf(myTraceSwitch.TraceInfo == true,
   "Type Mismatch Error");
```

Обратите внимание: операторы *Trace* не подключаются к трассировочным переключателям автоматически, за это целиком и полностью отвечает программист, то есть вы. При использовании методов, выполняющих безусловную запись, например *Trace.Write* или *Trace. WriteLine*, результаты трассировки записываются но всех случаях, независимо от состояния переключателей.

▶ Как создать трассировочные переключатели и задействовать их в коде

1. Создайте экземпляр класса *BooleanSwitch* или *TraceSwitch*, передав конструктору параметры *DisplayName* и *Description*;
2. При необходимости проверяйте состояние переключателей с помощью методов *Trace. WriteIf* и *Trace. WriteLineIf*.

Настройка трассировочных переключателей

Трассировочные переключатели разрешено включать и отключать в компилированном приложении через его конфигурационный файл — файл с расширением .config в формате XML, в котором хранится необходимая приложению информация. Конфигурационный файл располагается в одном каталоге с исполняемым файлом, а формат его имени — (*имя_приложения*).exe.config. Конфигурационные файлы есть не у всех приложений; если у вашего приложения, использующего трассировочные переключатели, такого файла нет, его необходимо создать. При исполнении кода, создающего трассировочный переключатель, приложение проверяет наличие в .config-файле сведений об этом переключателе, для каждого переключателя проверка производится однократно. Чтобы изменить параметры трассировочного переключателя после его создания, необходимо остановить приложение, внести соответствующие изменения в .config-файл и запустить приложение снова,

При создании трассировочного переключателя следует предоставить параметр *DisplayName*. который определяет имя переключателя в конфигурационном файле. При редактировании .config-файла следует указать имя переключателя и целочисленное значение, которое необходимо ему присвоить. Для объектов типа *BooleanSwitch* значение 0 деактивирует переключатель, а любые значения, отличные от нуля, активируют его. Для объектов *TraceSwitch* значения О, 1, 2, 3 и 4 представляют соответствен но уровни *TraceLevel. Off*, *TraceLevel. Error*, *TraceLevel.Warning*, *TraceLevel.Info* и *TraceLevel.Verbose,* любые значения больше 4 приравниваются к *TraceLevel. Verbose*.

Чтобы конечный пользователь смог при необходимости настроить трассировку программы, необходимо включить в .config-файл подробные комментарии с описанием назначения трассировочных переключателей.

Чтобы создать конфигурационный файл и настроить трассировочные переключатели, выполните следующие действия.

1. Создайте в программе объекты переключателей с соответствующими значениями свойства *DisplayName*.
1. Если у вашего приложения нет .config-файла, создайте его. Для этого:
   - Visual Basic .NET. Выберите в меню команду Project\Add New Item. В диалоговом окне Add New Item выберите шаблон Application Configuration File - откроется заготовка конфигурационного файла;
   - Visual C#. Выберите в меню команду Project\Add New Item. В диалоговом окне Add New Item выберите Text File, назовите новый файл **app.**config — откроется окно текстового редактора, где нужно ввести следующий XML-код:

```
<?xml version="1.0" encoding="Windows-1252"?>
<configuration>
</configuration>
```

3. Между тэгами <configuration> и </configuration> добавьте XML-код, объявляющий трассировочные переключатели и присваивающий им значения. Вот пример XML-кода для переключателей с именами *myBooleanSwitch* и *myTraceSwitch*:

```
<system.diagnostics>
   <switches>
      <add name="myBooleanSwitch" value="0" />
      <add name="myTraceSwitch" value="3" />
   </switches>
</system.diagnostics>
```

   В этом примере значение *myBooleanSwitch* установлено в *off*, а значение *myTraceSwitch* — в *TraceLevel.Info*.

4. Чтобы включить или выключить любой из переключателей, измените его значение. Например, для активизации переключателя *myBooleanSwitch* внесите в соответствующий элемент XML следующие изменения:

```
<add name="myBooleanSwitch" value="1" />
```

   Оставшийся XML-код оставьте без изменений.

5. При необходимости снабдите файл комментариями, описывающими назначение всех переключателей и значения их уровней. Это поможет пользователям понять, как работать с вашим приложением. Вот пример конфигурационного файла:

```
<system.diagnostics>
      <switches>
         <!- Это глобальный переключатель, активирующий трассировку,
              Чтобы включить трассировку, установите для него значение,
              отличное от нуля.
              Чтобы отключить трассировку, установите значение ноль ->
         <add name="myBooleanSwitch" value="0" />
         <!- Этот переключатель управляет уровнем трассировочных данных,
              Если трассировочные сведения не нужны, установите для него
              значение ноль.
              Значения 1,2,3 и 4 определяют вывод кратких, стандартных,
              подробных и развернутых сообщений соответственно ->
         <add name="myTraceSwitch" value="3" />
      </switches>
</system.diagnostics>
```

Резюме

- Классы *Debug* и *Trace* позволяют во время выполнения отображать и записывать в журнал сведения об условиях исполнения приложения, Эти классы функционально идентичны и предоставляют методы, генерирующие трассировочные данные.
- Данные, генерированные операторами *Trace* и *Debug,* передаются членам набора *Trace.Listeners*— объектам-слушателям, способным получать трассировочные данные и обрабатывать их. Существует три вида трассировочных слушателей:
  - *DefaultTraceListener*
  - *TextWriterTraceListener*
  - *EventLogTraceListener*
- Трассировочные переключатели служат для настройки операторов *Trace.* Объекты-переключатели *BooleanSwitch* могут быть включены либо отключены, а объекты *TraceSwitch* поддерживают свойство *TraceSwitch.Level,* позволяющее определить объем трассировочных данных;
- Трассировочными переключателями можно манипулировать и после компиляции приложения -- через конфигурационный файл приложения.

Занятие 3. Создание плана блочного тестирования

Написанный код приложения необходимо протестировать, чтобы удостовериться в его соответствии требованиям. Если синтаксические ошибки удается обнаружить при компиляции, то для выявления ошибок времени выполнения и логических ошибок необходимо тестирование. На этом занятии вы научитесь составлять эффективные планы для тестирования компонентов и приложений, а также научитесь вводить в приложение тестовые данные.

Изучив материал этого занятия, вы сможете:

- ✓ создать план тестирования классов;
- ✓ подготовить для тестирования приложения среду, поддерживающую несколько культур;
- ✓ загружать в приложение тестовые данные, соответствующие различным культурам.

Продолжительность занятия - около 30 минут.

План блочного тестирования

Редкая программа, только что вышедшая «из-под пера» разработчика, не содержит ошибок. Время от времени ошибаются даже весьма опытные программисты. Поэтому следует исходить из предположения, что любой только что написанный код содержит ошибки. Естественно, от приложения, которое кишит «жучками» и работает не так, как требуется, толку мало. Приложите все усилия, чтобы в окончательной версии приложения ошибок оказалось как можно меньше. Тестирование способно превратить полный ошибок опытный образец приложения в стабильную версию, пригодную для выпуска.

Тестирование и отладка — это два различных и в тоже время взаимосвязанных мероприятия. Под отладкой понимают непосредственный поиск ошибок в коде и их исправление, а тестированием называют процесс, позволяющий выявить эти ошибки. Обычно тестированию подвергают каждый метод приложения, заставляя его обработать всевозможные параметры в разных условиях. Такой подход называют *блочным тестированием*, поскольку при этом выполняется тестирование отдельных компонентов приложения. Однако приложения, как правило, слишком сложны, чтобы проверить все возможные параметры и условия исполнения. Рассмотрим следующий метод *DisplayName*:

Visual Basic .NET

```
Public Sub DisplayName(ByVal Name As String)
    System.Windows.Forms.MessageBox.Show(Name)
End Sub
```

Visual C#

```
public void DisplayName(string Name)
{
```

```
    System.Windows.Forms.MessageBox.Show(Name);
}
```

Предположим, что некоторый метод, вызывающий *DisplayName*, передает в параметре *Name* строку длиной до восьми символов стандартного английского алфавита. Это означает, что метод из нашего примера может получить 26^8 или примерно 200 миллиардов различных строк. Ясно, что протестировать их все в данном случае невозможно.

Но столь строгое тестирование здесь и не требуется. В случае с предыдущим примером мало вероятно, что различные строки будут вызывать различные ошибки. А значит, для проверки работоспособности такого метода достаточно протестировать его на нескольких представительных значениях аргументов. Подобные примеры называются *контрольными примерами* (test cases).

Успех тестирования целиком определяется качественным выбором контрольных примеров. Если их слишком мало, тестирование не даст результата и в окончательной версии программы непременно окажется много ошибок, а если их чрезвычайно много, вы потеряете время, деньги и превысите сроки, отведенные на разработку. Сбалансированным планом тестирования приложения считается тот, где достаточно полно представлены функции приложения и проверяются наиболее типичные сценарии его использования.

Разработка контрольных примеров

Как говорилось ранее, следует исходить из того, что проверке подлежит каждая строка кода. Следовательно, если в тестируемом методе есть структуры решений, потребуется контрольный пример для проверки всех вариантов поведения этой структуры. Рассмотрим *следующий* метод:

Visual Basic .NET

```
Public Sub TestMethod(ByVal X As Boolean, ByVal Y As Boolean)
   If X = True Then
      MessageBox.Show("X is True")
   Else
      MessageBox. ShowC'X is False.")
   End If
   If Y = True Then
      MessageBox.Show("Y is True")
   Else
      MessageBox. Show("Y is False")
   End If
End Sub
```

Visual C#

```
public void TestMethod(bool X, bool Y)
{
   if (x == true)
      MessageBox.Show("X is true");
   else
      MessageBox.Show("X is false");
```

```
    if (Y == true)
        MessageBox.Show("Y is true");
    else
        MessageBox.Show("Y is false");
}
```

Для проверки всех строк этого метода достаточно двух контрольных примеров: в первом значения *X* и *Y* должны быть равны *true*, а во втором — *false*. Эти примеры позволят задействовать все строки кода этого метода.

Однако, по некотором размышлении становится ясно, что этот подход не обеспечивает необходимого охвата функциональности. Действительно, задействованы все строки кода, но примеры статичны: они не обеспечивают тестирование с переменным набором данных. Нашему методу могут быть переданы параметры, которые направят его исполнение по пути, не предусмотренному контрольным примером (например, *X= true* и *Y = false)*. Чтобы рассмотреть все возможные варианты при тестировании этого метода, потребуется два дополнительных контрольных примера: в первом *X= true*, а *Y =false,* а во втором — *X = false*,а *¥= true.* Тестирование всех возможных путей передачи данных составляет программу-минимум при разработке плана блочного тестирования.

Составление тестовых данных

Проверка функциональности программы при обработке всех вариантов однотипных данных — хорошая отправная точка для составления плана тестирования. Но только тестирование обработки разных типов данных позволяет убедиться в надежности программы. Приложение должно не только нормально работать и генерировать ожидаемые результаты на основе аргументов, на работу с которыми оно рассчитано, но и корректно обрабатывать значения, которые выходят за пределы допустимого диапазона. Тестирование считается законченным не раньше, чем будет установлено, что оно корректно обрабатывает различные данные, в том числе и значения, которые больше или меньше допустимых. Далее описаны приемы составления тестовых данных, которые пригодятся вам при создании контрольных примеров для тестирования собственных приложений.

Проверка типичных значений аргументов

Важно убедиться, что программа правильно обрабатывает данные, с которыми ей придется сталкиваться чаше всего. В большинстве случаев проверить при тестировании все значения этого диапазона невозможно, однако в контрольных примерах должно быть несколько вариантов типичных данных, которые будет обрабатывать приложение.

Проверка обработки минимальных и максимальных значений аргументов

Особое внимание необходимо уделять проверке границ допустимого диапазона аргументов. С этой целью в контрольные примеры включают не только крайние значения из этого диапазона, но и аргументы, которые на единицу больше и меньше минимального и максимального значений. Например, чтобы протестировать верхнюю границу диапазона аргументов, необходимо проверить три значения: максимальное допустимое значение, а также значения на единицу больше и на единицу меньше его. Это позволит «отловить» ряд стандартных ошибок, например использование оператора > вместо >=.

Использование заведомо недопустимых аргументов

Таким способом удается удостовериться, что ввод подобных данных не вызовет крах приложений или, хуже того, что программа не будет возвращать некорректные результаты, хотя внешне ее работа кажется нормальной. Для этого необходимо включить в контрольные примеры аргументы, значения которых выходят за пределы допустимого диапазона. Например, передать ноль в параметре, в котором ноль передавать нельзя; отрицательное число вместо положительного и т.п.

Комбинированные примеры

Рекомендуется включать в контрольные примеры различные комбинации упомянутых тестовых данных. Некоторые ошибки могут не проявиться, пока приложение получает корректные комбинации данных. Например, метод функционирует правильно, если его параметрам *поочередно* передаются максимальные допустимые значения, но генерирует ошибку, если максимальные значения передаются всем его параметрам *сразу*. И наоборот: определенные комбинации заведомо недопустимых данных генерируют внешне нормальные результаты. Рассмотрим следующий метод:

Visual Basic .NET

```
Public Function CalculateMySalary(ByVal Hours As Short, ByVal Weeks _
   As Short) As Integer
   ' Этот метод предполагает наличие переменной WageConstant,
   ' представляющую размер почасовой оплаты.
   Dim mySalary As Integer
   mySalary = Hours * Weeks * WageConstant
   Return mySalary
End Function
```

Visual C#

```
public int CalculateMySalary(short Hours, short Weeks)
{
   // Этот метод предполагает наличие переменной WageConstant,
   // представляющую почасовую оплату.
   int mySalary;
   mySalary = Hours * Weeks * WageConstant;
   return mySalary;
}
```

Этот метод нормально работает, если ему передаются допустимые аргументы: значение *Hours* из диапазона 0—40, а также число недель от 0 до 52. Но что будет, если вызвать его так:

Visual Basic .NET

```
Dim Hooray As Integer
Hooray = CalculateMySalary(-40, -52)
```

Visual C#

```
int Hooray;
Hooray = CalculateMySalary(-40, -52);
```

В этом случае метод, принимая комбинацию заведомо неверных данных, возвращает внешне совершенно нормальный результат. Вывод очевиден, необходимо разрабатывать контрольные примеры так, чтобы проверить различные комбинации данных.

План блочного тестирования создают следующим образом.

1. Прежде всего создают контрольные примеры, которые позволят проверить каждую строку тестируемого блока кода.
2. Далее разрабатывают контрольные примеры для тестирования всех вариантов обработки данных тестируемым блоком.
3. Затем пишут контрольные примеры, которые позволят протестировать различные виды данных, обрабатываемых данным элементом. Помимо тестирования обработки типичных значений необходимо уделить внимание проверке минимальных и максимальных значений аргументов, обработке неверных данных и их комбинадий, а также допустимых значений, в том числе минимальных л максимальных.
4. Прежде чем приступать к тестированию, рассчитайте ожидаемые результаты для всех контрольных примеров.
5. Выполните тесты и сравните полученные результаты с ожидаемыми.

Резюме

- Блочное тестирование позволяет проверить все компоненты приложения, чтобы удостоверится, что они работают так, как предписано. Блочное тестирование выполняется с использованием контрольных примеров. Контрольный пример представляет собой набор условий периода выполнения, обеспечивающих проверку определенного метода обработки данных и/или комбинации значений аргументов.
- Контрольные примеры должны обеспечивать проверку как можно большего числа вариантов обработки данных и комбинаций аргументов, то есть всех вариантов обработки данных тестируемым блоком кода, Дополнительно следует тестировать обработку:
  - типичных данных;
  - максимальных и минимальных допустимых значений аргументов;
  - недопустимых данных;
  - а также комбинаций перечисленных выше вариантов.

Занятие 4. Обработка и генерация исключений

Даже тестирование и отладка приложения не исключают возможность возникновения ошибок в период выполнения. Рассмотрим для примера приложение, записывающее данные на гибкий диск. Если оно попытается сохранить данные, когда диск не готов, непременно произойдет ошибка, даже если код приложения сам по себе не содержит ошибок и работает, как задумано. Языки Visual Basic .NET и Visual C# поддерживают структурную обработку исключений, которая позволяет создавать код, способный корректно обрабатывать ошибки, и значит, продолжить выполнение программы. Кроме того, можно создавать объекты исключений и генерировать исключения внутри компонентов для уведомления основной программы о возникновении ошибки.

Изучив материал этого занятия, вы сможете:

- ✓ описать обработку исключений общеязыковой исполняющей средой;
- ✓ создать структурный обработчик исключения;
- ✓ генерировать исключения в программе.

Продолжительность занятия - **около** 30 минут.

Принципы обработки исключений

Обнаружив ошибку периода выполнения, приложение генерирует *исключение* (exception). Исключения представляют собой экземпляры специализированных классов, потомков класса *System.Exception*. Эти классы поддерживают функции, облегающие диагностику ошибок и управление их обработкой, например:

- свойство *Message* — содержит описание ошибки в доступной для человека форме и другие важные сведения об ошибке;
- свойство *StackTrace* — хранит данные трассировки стека; обеспечивает трассировку для поиска места в коде, где возникла ошибка.

В дополнение к исключениям, определяемым системой, разрешается создавать пользовательские исключения на основе класса *System.ApplicationException*Подробнее об этом — в следующих разделах.

При возникновении ошибки периода выполнения создается экземпляр соответствующего исключения и передается от метода, генерировавшего ошибку, вверх по стеку вызовов, В результате исключение получает метод, из которого был вызван тот, что сгенерировал ошибку. Если в этом методе имеется структура, способная обработать это исключение, оно обрабатывается; в противном случае оно передается следующему методу в стеке вызовов и т.д. Если структура, обрабатывающая исключения, не найдена в стеке вызова, вызывается обработчик исключений, заданный по умолчанию. Он отображает сообщение с указанием типа исключения и любых дополнительных сведений, предоставленных объектом исключения, и закрывает приложение, не позволяя сохранить результаты работы или исправить ошибку.

Применение структурной обработки исключений позволяет приложению устранить последствия непредвиденных ошибок или, как минимум, сохранить данные перед закрытием.

Создание **обработчиков** исключений

Обработчики исключений реализуют для отдельных методов. У каждого метода должен быть собственный обработчик исключений, ориентированный на особенности данного метода и приспособленный для обработки исключений, которые он обычно генерирует. Обработку исключений непременно следует реализовать в методах, которые часто генерируют исключения, например в использующих файловый доступ. Для обработчиков исключений принят синтаксис *Try... Catch... Finally.* Вот основные этапы создания обработчика исключений:

1. поместите код, способный генерировать исключения, в блок *Try (try);*
2. добавьте один или несколько блоков *Catch (catch),* обрабатывающих различные типы исключений, которые может генерировать код в блоке *Try (try);*
3. поместите в блок *Finally (finally)* код, который необходимо исполнить в любом случае. Этот код будет исполнен независимо от того, возникло ли исключение.

Создание обработчика начинается с заключения некоторого кода в блок *Try (try),* формирующий обработчик исключений. Если при исполнении любого из операторов, заключенных в блок *Try (try),* возникнет исключение либо необработанное исключение будет передано из стека, метод попытается обработать его с помощью данного обработчика. Если это ему не удастся, исключение передается вверх по стеку вызовов. Вот пример блока *Try (fry):*

Visual Basic .NET

```
Try
    ' Здесь находится обычный код приложения,
    ' а сюда помещают код для обработки ошибок, например блоки Catch
    ' и Finally; оператор End Try завершает блок для обработки ошибок,
End Try
```

Visual C#

```
try
{
   // Здесь находится обычный код приложения,
}
   // а код для обработки ошибок, например блоки Catch и Finally,
   // размещают вне фигурных скобок, отделяющих блок try.
```

В Visual Basic .NET из блока *Try* выходят посредством оператора *Exit Try.*

За блоком *Try (try)* может следовать один или несколько блоков *Catch (catch),* где находится код, который должен быть исполнен при обработке исключения. Блоки *Catch (catch)* бывают универсальными, перехватывающими любые исключения, либо специализированными, которые обрабатывают только определенный класс исключений, Ниже показан пример универсального блока *Catch (catch),* перехватывающего все исключения, генерированные в соответствующем блоке *Try (try):*

Visual Basic .NET

```
' Здесь находится блок Try
Catch
' Блок Catch завершается оператором End Try, другим блоком Catch
```

или блоком Finally. Код, размещенный в этих блоках, будет выполнен при возникновении ошибок периода выполнения.

Visual C#

```
// Здесь находиться блок Try
catch
{
   // Блок catch ограничен фигурными скобками. Код, заключенный между ними,
   // будет исполнен при возникновении ошибок периода выполнения,
}
```

Оператор *Catch (catch)* также позволяет получить ссылку на объект генерированного исключения. Эта ссылка полезна, так как в обработчике ошибок через нее удается получить доступ к объекту исключения. Кроме того, она позволяет перехватывать определенные типы ошибок и создавать несколько блоков *Catch* (*catch*), предназначенных для обработки определенных ошибок, Так, например, можно перехватывать исключение определенного типа:

Visual Basic .NET

```
' Здесь находиться блок Try
Catch e As System.NullReferenceExceptior
    ' Если в блоке Try будет генерировано исключение NullReferenceException,
    ' этот блок Catch перехватит его. При этом в переменной e будет ссылка
    ' на объект исключения, открывающая доступ к сведениям об исключении.
Catch e As System.Exception
    ' Поскольку все исключения являются производными от класса
    ' System.Exception, этот блок Catch будет перехватывать любые исключения,
    ' не перехваченные предыдущим блоком Catch.
```

Visual C#

```
// Здесь находиться блок Try
catch (System.NullReferenceException e)
{
   // Если в блоке Try будет генерировано исключение NullReferenceException,
   // этот блок Catch перехватит его. Пои этом в переменной e будет ссылка
   // на объект исключения, открывающая доступ к сеедениям об исключении.
}
catch (System.Exception e)
{
   // Поскольку все исключения являются производными от класса
   // System.Exception, этот блок Catch будет перехватывать любые исключения,
   // не перехваченные предыдущим блоком Catch.
}
```

В блок *Finally* (*finally*) помещают код, который необходимо исполнить независимо от того, возникло исключение или нет. Это может быть код, сохраняющий данные, освобождающий ресурсы или код, исполнение которого критично для приложения. Вот пример блока *Try...Catch...Finally*(*try...catch...finally*):

Visual Basic .NET

```
Public Sub Parse(ByVal aString As String)
   Try
      Dim aDouble As Double
      ' Если aString - Nothing, генерируется исключение ArgumentNullException
      aDouble - Double.Parse(aString)
   Catch e As System.ArgumentNullException
      ' Здесь можно разместить код, который должен быть исполнен при
      ' возникновении исключения System.ArgumentNullException.
   Catch e As System.Exception
      ' Этот блок перехватывает все исключения, которые не были
      ' перехвачены предыдущим блоком.
   Finally
      ' Сюда помещают код, который необходимо исполнить
      ' независимо от того, возникло ли исключение или нет.
   End Try
End Sub
```

Visual C#

```
public void Parse(string aString)
{
   try
   {
      double aDouble;
      aDouble = Double.Parse(aString);
   }
   catch (System.ArgumentNullException e)
   {
      // Здесь можно разместить код, который должен быть исполнен при
      // возникновении исключения System.ArgumentNullException.
   }
   catch (System.Exception e)
   <
      // Этот блок перехватывает все исключения, которые не были
      // перехвачены предыдущим блоком.
   }
   finally
   {
      // Сюда помещают код, который необходимо исполнить
      // независимо от того, возникло ли исключение или нет.
   }
}
```

Если при исполнении кода, заключенного в блок *Try (try),* **возникает ошибка, управление немедленно** передается **блокам** *Catch* (*catch*). Далее **все блоки** ***Catch*** *(catch)* проверяются **в поисках того, что перехватывает** данное **исключение.** Ранее показаны **два блока** ***Catch*** (*catch*): один **перехватывает только одно** исключение

(*ArgumentNullException*)а второй — все остальные. При возникновении ошибки исполняется только один блок *Catch (catch);* после исполнения кола в этом блоке управление получает код из блока *Finally* (*finally*).Далее возобновляется нормальное исполнение приложения со строки, следующей за строкой, откуда вызвана процедура, генерировавшая ошибку.

Примечание Поскольку при обработке любого исключения исполняется только один блок *Catch (catch)*, необходимо разместить блоки *Catch (catch)* так, чтобы первым шел блок, перехватывающий наиболее специфичный тип исключений, а последним — блок, перехватывающий самый общий тип исключений. Таким образом, сначала следует разместить в коде блоки *Catch (catch),* обрабатывающие более специфичные типы исключений, а после них — универсальную процедуру обработки ошибок.

Иногда обрабатывать исключение при помощи локального обработчика нежелательно, вместо этого лучше передать его по стеку методу, вызвавшему процедуру, при исполнении которой возникла ошибка. В этом случае тоже следует исполнить некоторый код при возникновении исключения. В подобной ситуации разрешено опустить блок *Catch (catch)* и использовать структуру *Try...Finally*(*try...finally*)следующим образом:

Visual Basic .NET

```
Try
    ' Здесь находится код метода.
Finally
    ' Здесь находится код, выполняющий очистку.
End Try
```

Visual C#

```
try
{
   // Здесь находится код метода.
}
finally
{
   // Здесь находится код, выполняющий очистку.
}
```

▶ **Создание обработчика исключений**

1. Разместите в блоке *Try (try)* код, способный генерировать ошибки, которые нужно обработать.
2. Если исключения должны обрабатываться локально, создайте один или несколько блоков *Catch (catch)* с соответствующим кодом.
3. Код, который необходимо исполнить независимо от возникновения исключения, поместите в блок *Finally* (*finally*).

Генерация исключений

Весьма часто требуется генерировать исключения. В общем случае их можно подразделить на два типа:

- при возникновении ошибки периода выполнения, которую не удается обработать полностью, После частичной обработки такой ошибки следует генерировать исключение, чтобы передать ее по стеку для дальнейшей обработки;
- при работе компонента возникла недопустимая ситуация, которую не удается разрешить локально. Об этом необходимо уведомить родительский компонент, сгенерировав стандартное исключение .NET либо нестандартное исключение, специфичное для данного компонента.

Оба данных случая подробно рассматриваются в этом разделе.

Повторная генерация исключений

Иногда структуре обработки исключений не удается полностью обработать возникшее исключение. Например, во время работы метода, выполняющего множество сложных вычислений, иногда неожиданно возникает исключение *Null ReferenceException*. В этом случае успешная обработка исключения возможна, вероятен также исход, когда это не удается. Ваш код должен проверить, в состоянии ли он обработать возникшее исключение. Если это не удается в контексте блока *Catch* (*catch*), следует генерировать его повторно при помощи ключевого слова *Throw* (*throw*), например:

Visual Basic .NET

```
Try
    ' Здесь должен быть код приложения,
Catch e As System.NullReferenceException
    ' а сюда помещают код, который проверяет исключение и текущий контекст,
    ' чтобы определить, можно ли обработать исключение в данной блоке Catch,
    ' или его нужно передать дальше по стеку. Если исключение нельзя
    ' обработать локально, следующая строка генерирует исключение повторно
    ' и передает его дальше по стеку.
   Throw e
End Try
```

Visual C#

```
try
{
   // Здесь должен быть код приложения,
}
catch (System.NullReferenceException e)
{
   // а сюда помещают код, который проверяет исключение и текущий контекст,
   // чтобы определить, можно ли обработать исключение в данном блоке Catch,
   // передано выше по стеку. Если оно не может быть обработано локально.
   // локально, следующая строка генерирует исключение повторно и передает
   // его дальше по стеку.
```

```
    throw e;
}
```

При использовании универсального блока *Catch (catch),* не предоставляющего ссылку на объект исключения, полученное этим блоком исключение можно генерировать повторно посредством ключевого слова *Throw* (*throw*), указав его без аргументов;

Visual Basic .NET

```
' Это начало блока Catch.
Catch
    Throw
```

Visual C#

```
// Это начало блока Catch.
catch
{
    throw;
}
```

В некоторых случаях необходимо передать на следующий уровень приложения дополнительные сведения. Для этого вместо повторной генерации возникшего исключения следует генерировать другое исключение, в этом случае второе исключение играет роль оболочки для первого. Ссылка на объект исходного исключения записывается в свойство *InnerException*объекта нового исключения его конструктором. Этот прием позволяет передать с системным типом исключения дополнительные данные, записав их в соответствующие свойства нового исключения и поместив исходное исключение в свойство *Inner Exception.* Ниже демонстрируется создание и генерация нового исключения, предоставляющего строку с подробным описанием исходного исключения, оболочкой которого оно является:

Visual Basic .NET

```
' Это начало блока Catch.
Catch e as NullReferenceException
    Throw New NullReferenceException( "Widget A is not set", e)
```

Visual C#

```
// Это начало блока Catch.
catch (NullReferenceException e)
{
    throw new NullReferenceException("Widget A is not set", e);
}
```

▶ **Как повторно генерировать исключение**

- чтобы повторно генерировать исключение, полученное блоком *Catch (catch),* воспользуйтесь ключевым словом *Throw (throw);*
- чтобы генерировать новое исключение, предоставляющее дополнительные сведения об исходном исключении, создайте объект исключения, являющийся обо-

лочкой для исходного исключения, и генерируйте новое исключение с помощью ключевого слова *Throw (throw)*.

Нестандартные исключения

При разработке компонентов иногда также требуется генерировать исключения. Если во время работы компонента возникает недопустимая ситуация, следует генерировать исключение для клиентского приложения. Для этих целей всегда используют нестандартные исключения, поскольку общеязыковая исполняющая среда генерирует исключение лишь в предопределенных исключительных ситуациях,

Примечание Согласно названию, исключения предназначены лишь для нестандартных ситуаций. Не следует использовать их для взаимодействия между клиентом и компонентами — для этого предназначены события. Исключения следует генерировать, только когда продолжение нормального исполнения программы без внешнего вмешательства невозможно.

Нестандартные исключения можно создавать на основе класса *System.ApplicationException*. Это класс инкапсулирует всю необходимую функциональность, в том числе свойства *Message*, *StackTrace* и *InnerException*, и может стать основой для нестандартных исключений, ориентированных на потребности вашего компонента. Ниже показан пример нестандартного исключения, созданного на основе класса *ApplicationException*. Конструктор нового исключения требует фиктивный объект *Widget*, поскольку новое исключение предоставляет доступ к этому объекту через свойство, доступное только для чтения.

Visual **Basic .NET**

```
Public Class WidgetException
   Inherits System.ApplicationException
   ' Эта переменная хранит объект Widget.
   Private mWidget As Widget.
   ' Это свойство служит для доступа к объекту Widget.
   Public Readonly Property ErrorWidget() As Widget
      Get
         Return mWidget
      End Get
   End Property
   ' Этот конструктор принимает объект Widget и строку String, эти
   ' параметры можно использовать для описания контекста приложения
   ' на момент ошибки. Разрешается также создавать перегруженные версии
   ' конструктора, поддерживающие различные наборы параметров.
   Public Sub New(ByVal W As Widget, ByVal Message As String)
      ' Следующая строка вызывает конструктор из базового класса
      ' и устанавливает свойство Message,
      ' унаследозанное от класса ApplicationException.
      MyBase.New(Message)
      ' Установить свойство Widget,
      mWidget - W
```

```
    End Sub
End Class
```

Visual C#

```
public class WidgetException:System.ApplicationException
{
    // Эта переменная хранит объект Widget.
    Widget mWidget;
    public Widget ErrorWidget
    {
        get
        {
            return mWidget;
        }
    }
    // Этот конструктор принимает объект Widget и строку String, эти
    // параметры можно использовать для описания контекста приложения
    // на момент ошибки. Разрешается также создавать перегруженные версии
    // конструктора, поддерживающие различные наборы параметров.
    //
    // Следующая строка вызывает конструктор из базового класса
    // и устанавливает свойство Message,
    // унаследованное от класса ApplicationException.
    public WidgetException(Widget W. string S) : base(S)
    !
        // Установить свойство Widget.
        mWidget = W;
    }
}
```

Объявив класс исключения, удается сгенерировать новый экземпляр этого класса при возникновении недопустимой ситуации, например, так:

Visual Basic .NET

```
Dim Alpha As New Widget ()
' Здесь должен быть код, повреждающий объект Widget с именем Alpha.
Throw New WidgetException(Alpha, "Widget Alpha is corrupt!")
```

Visual C#

```
Widget Alpha;
// Здесь должен быть код, повреждающий объект Widget с именем Alpha.
throw new WidgetException(Alpha, "Widget Alpha is corrupt!");
```

▶ **Как генерировать нестандартное исключение**

1. Объявите класс нестандартного исключения. Он должен быть потомком *System.ApplicationException* и может содержать любые свойства, предоставляющие дополнительные данные клиентам, обрабатывающим исключения. Эти свойства инициализируют в конструкторе класса нестандартного исключения.

2. При возникновении нештатной ситуации генерируйте новый экземпляр класса нестандартного исключения при помощи ключевого слова *Throw (throw)*, передав конструктору соответствующие параметры.

Резюме

- При помощи исключений общеязыковая исполняющая среда уведомляет об ошибках периода выполнения. Генерированное исключение передается вверх по стеку, пока не встретится структура, способная его обработать. Если таковая не найдена, общеязыковая исполняющая среда сама обрабатывает это исключение при помощи процедуры обработки исключений по умолчанию, которая прекращает исполнение программы.
- Разработчик может написать собственные процедуры для обработки исключений, позволяющие приложению устранить последствия неожиданной ошибки или по крайней мере корректно завершить свою работу. Как правило, структура обработки исключений состоит из трех блоков;
  - *Try (try);*
  - *Catch (catch);*
  - *Finally (finally).*
- После обработки исключения управление возвращается методу, в котором находится процедура, обработавшая исключение.
- Ключевое слово *Throw (throw)* применяют в блоке *Catch (catch)* для повторной генерации обрабатываемого исключения. Также можно создать новое исключение, которое станет оболочкой для исходного исключения, при этом оно записывается в свойство *InnerException* объекта нового исключения.
- Для объявления и генерации нестандартных исключений используют класс *System.ApplicationException*. Не следует применять исключения для взаимодействия между компонентами и их клиентами; они служат только для разрешения исключительных ситуаций.

Практикум 5. Отладка приложения

Вы запустите приложение, содержащее синтаксические и логические ошибки, а также ошибки периода выполнения. Ваша задача — найти и исправить синтаксические ошибки, а также разработать план тестирования, который позволит обнаружить и устранить ошибки периода выполнения и логические ошибки в двух методах. Для наблюдения за работой программы вам понадобятся отладочные инструменты из Visual Studio. Решение задач этого практикума находится в папке \Labs\Ch05\Solution на прилагаемом к книге компакт-диске.

Прежде всего

Этот практикум не требует специальной подготовки.

Продолжительность практикума - около 60 минут.

Упражнение 1. Поиск синтаксических ошибок

С помощью Visual Studio .NET вам следует найти и исправить синтаксические ошибки в предложенном проекте. Прежде всего откройте проект Lab 5.

1. Запустите Visual Studio.
2. В меню File выберите команду Open\Project — откроется диалоговое окно Open Project.
3. Откройте папку \Labs\Ch05\Partial и выберите решение VB либо C Sharp. Щелкните кнопку Open, чтобы открыть выбранное решение.

Чтобы найти и исправить синтаксические ошибки, сделайте следующее.

1. Выберите в меню Build команду Build Solution — в окне Task List (рис. 5-3 и 5-4) появится список синтаксических ошибок.

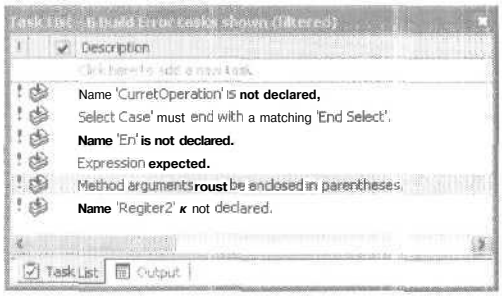


Рис. 5-3. Окно Task List для кода на Visual Basic .NET

2. **Visual Basic .NET**: дважды щелкните ошибку с описанием «Select Case' must end with a matching End Select» — курсор будет перемещен в конец блока Select Case. Проверьте код и исправьте ошибку в строке End Select. Обратите внимание: в окне Task List остались только две ошибки.

   Visual C#: дважды щелкните ошибку с описанием «} expected» — курсор будет перемешен в конец обработчика операции. Тщательно проверьте все фигурные

скобки, чтобы найти незакрытую. Исправьте ошибку, закрыв фигурную скобку после оператора *switch*.

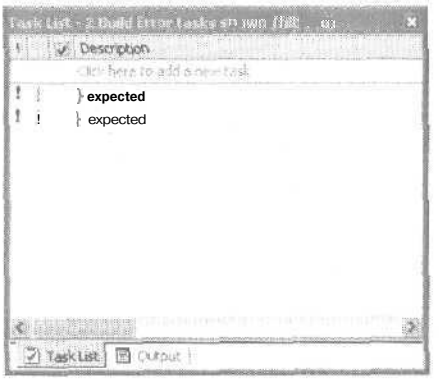


Рис. 5-4. **Окно Task List для кода на Visual C#**

3. **Только для Visual** C#: в меню Build выберите команду Build Solution, чтобы проверить код еще раз. Заметьте, что найдено еще две ошибки.
4. Дважды щелкните первую ошибку в списке Task List — курсор будет перемещен в строку, генерирующую ошибку из-за невозможности найти член *CurretOperation*. Исправьте оператор: правильное имя — *CurrentOperation*.
5. Дважды щелкните последнюю ошибку в списке Task List. Проверьте, нет ли опечаток в имени переменной *Register2*. Исправьте ошибку.
6. В меню Build выберите команду Build Solution — ошибок больше не обнаружено.

Исправив все синтаксические ошибки, перейдите к поиску ошибок периода выполнения и логических ошибок. Протестируйте два метода приложения-примера, содержащие ошибки. Изучите все методы и разработайте план их тестирования, предусматривающий проверку всех путей обработки данных путем передачи методу допустимых и недопустимых аргументов. Протестируйте метод согласно составленному плану. Закончив, сравните свой результат с готовым решением из капки \Labs\Ch05\Solution.

▶ **Как найти и исправить ошибку периода выполнения**

1. Найдите метод *NumberHandler*— он довольно простой и почти не использует ветвление, Это метод отображает цифры, соответствующие нажатым кнопкам калькулятора. Далее внесите в код исправления:

Visual Basic .NET

```
CurrentNumberString = CurrentNumberString & CType((CType(sender,
Textbox).Tag), String)
```

замените на:

```
CurrentNumberString = CurrentNumberString & CType((CType(sender,
Button).Tag), String)
```

Visual C#

```
CurrentNurnberString = CurrentNumberString +
((TextBox)sender).Tag.ToString();
```

замените на:

```
CurrentNumberString = CurrentNumberString +
((Button)sender).Tag.ToString();
```

2. Составьте план для его тестирования. Ваш план должен проверять все пути обработки данных, а также обработку различных входных данных и аргументов, переданных методу программно. Заранее рассчитайте верные результаты для каждого контрольного примера. Несколько контрольных примеров вы найдете в таблице 5-4.

Таблица 5-4. Контрольные примеры для тестирования метода *NumberHandler*

Кнопка	Значение *LabelDisplay. Text*	Фактическое значение *CurrentNumberString*	Ожидаемое значение *CurrentNumberString*
1	**0**	**Пустая строка**	1
4	14	**44**	**444**
y	**1266345578456**	**1266345578456**	**12663455784569**
9	-3446.222231	**-3446.222231**	**-3446.2222319**
3	**6**	**6**	**63**

3. Выполните запланированные тесты. Для наблюдения за значениями переменных во время исполнения программы пользуйтесь окнами Locals, Autos и Watch, для назначения значений переменным — окном Immediate. Для повторного исполнения кода воспользуйтесь командой Set Next Statement.
4. Запишите результаты и исправьте все обнаруженные ошибки периода выполнения.

Труднее всего диагностировать логические ошибки. Часто для выявления источника ошибки приходится проводить тестирование многократно, а также использовать дополнительные тесты. Ваша задача — найти логическую ошибку в одном из методов приложения.

▶ **Как найти и исправить** логическую **ошибку**

1. Найдите метод *EqualsHandler*— при нажатии кнопки «=» он выполняет сложение, вычитание, умножение или деление чисел, содержащихся в переменных *Registerl* и *Register2,* присваивает результат переменной *Result* и обновляет данные, отображаемые пользовательским интерфейсом.
2. Выполните тесты, описанные в таблице 5-5, — это далеко не полный план тестирования (анализ крайних значений аргументов и дополнительные тесты для краткости опущены), но указанных процедур вполне достаточно для проверки всех логических путей метода и выявления логической ошибки.

Таблица 5-5. Контрольные примеры для тестирования метода *EqualsHandler*

Операция	Registerl	Register2	Ожидаемое значение Result
Operation. Plus	442	5789	6231
Operation. Minus	400000	222222	177778
Operation.Multiply	23	12	276
Operation.Divide	3345	34	98.3823529411765

3. Найдите причины всех выявленных логических ошибок и устраните их.

Закрепление материала

Приведенные ниже вопросы помогут вам лучше усвоить основные темы данной главы. Если вы не сумеете ответить на вопрос, повторите материал соответствующего занятия. Ответы для самопроверки — в приложении «Вопросы и ответы» в конце книги.

1. Опишите режим пошагового исполнения кода и доступные в нем возможности.

2. Для чего служит окно Watch?

3. Представьте, что вы тестируете бета-версию большого приложения. Вам нужно регистрировать во время выполнения сведения о производительности приложения в текстовых файлах. Кратко опишите стратегию решения этой задачи.

4. Нужно ли при тестировании метода проверять обработку заведомо недопустимых аргументов? Почему?

5. Кратко опишите назначение каждой части конструкции *Try...Catch...Finally* (*try...catch...finally*).

ГЛАВА 6

Доступ к данным с помощью ADO.NET

Занятие 1. Обзор ADO.NET 236

Занятие 2. Доступ к данным 240

Занятие 3. Применение объектов DataSet и обновление данных 262

Занятие 4. Связывание, просмотр и фильтрация данных 278

Занятие 5. Применение XML в ADO.NET 291

В этой главе

Эта глава посвящена организации доступа к данным с помощью ADO.NET. **Вы** узнаете, как дополнить свое приложение архитектурой доступа к данным, научитесь подключаться к SQL Server и другим базам данных (БД), связывать данные с элементами пользовательского интерфейса и обрабатывать ошибки, возникающие при работе с данными.

Прежде **всего**

Для занятий по материалам этой главы необходимо выполнить занятия из глав 1-5.

Занятие 1. Обзор ADO.NET

Большинству приложений требуется для работы тот или иной тип доступа к данным. Клиентским приложениям необходимо взаимодействовать с централизованной БД, хранилищами данных в формате XML или локальными БД, работающими на клиентских машинах. Технология ADO.NET предоставляет простые в применении, но мощные средства доступа к данным, которые позволяют максимально полно задействовать ресурсы системы.

Изучив материал **этого** занятия, вы сможете:

- ✓ описать основные компоненты доступа к данным ADO.NET;
- ✓ объяснить роль каждого из этих компонентов;
- ✓ кратко описать организацию доступа к данным ADO.NET.

Продолжительность занятия - около 30 минут.

Различные приложения выдвигают разные требования к организации доступа к данным. Неважно, что должно делать приложение: отображать содержимое таблиц или обрабатывать и обновлять данные на центральном сервере SQL, — ADO.NET предоставляет разработчику простые в применении и эффективные инструменты, позволяющие реализовать доступ к данным в любых сценариях.

Доступ к отсоединенным данным

Прежние технологии доступа к данным по умолчанию обеспечивали доступ к данным через постоянное соединение с источником. В подобной модели приложение открывает соединение с БД и не закрывает его до завершения работы приложения или по крайней мере до завершения работы с источником данных. По мере роста сложности приложений число клиентов, обслуживаемых БД, неуклонно возрастает, при этом технология доступа к данным, использующая постоянное соединение, становится неудобной в силу следующих причин:

- поддержание соединения с БД «накладно» с точки зрения использования системных ресурсов: чем больше открытых соединений приходится поддерживать, тем ниже производительность системы;
- приложения, использующие доступ к данным через постоянное соединение, очень плохо масштабируются. Такое приложение хорошо обслуживает соединения с двумя клиентами, с трудом справляется с 10 и совершенно не годится для 100.

В ADO.NET эти проблемы решаются использованием по умолчанию модели доступа на основе отсоединенных данных. В этой модели соединение с источником данных открыто только до завершения необходимых действий над данными. Например, если приложение запрашивает данные из БД, соединение устанавливается только на время загрузки данных, после чего сразу же закрывается, Аналогично при обновлении БД соединение открывается на время исполнения команды UPDATE, а затем закрывается. Поддерживая соединения открытыми в течение минимально необходимого времени, ADO.NET экономно использует системные ресурсы и позволяет масштабировать инфраструктуру доступа к данным —производительность снижается при этом незначительно.

Архитектура данных ADO.NET

Доступ к данным в ADO.NET основан на использовании двух компонентов: *набора данных* (он представлен объектом *DataSet*), в котором данные хранятся на локальном компьютере, и *провайдера данных* (Data Provider), выполняющего функции посредника при взаимодействии программы и БД.

Объект DataSet

Объект *DataSet* — это представление в памяти компьютера данных, изолированных от источника данных. Этот объект можно также рассматривать как локальную копию фрагмента БД. В *DataSet* данные можно загрузить из любого допустимого источника, например из БД SQL Server, Microsoft Access или XML-файла. Объект *DataSet* хранится в памяти, его содержимым разрешено манипулировать и обновлять независимо от БД, играющей роль источника данных. При необходимости объект *DataSet* может служить шаблоном для обновления серверной БД.

Объект *DataSet* содержит набор объектов *DataTable* (этот набор может быть и пустым, то есть не содержать ни одного объекта *DataTable*). Каждый объект *DataTable* представляет в памяти компьютера одну таблицу. Структура объекта *Data Table* определяется двумя наборами: *DataColumns*, куда входят все столбцы таблицы, которую представляет объект *DataTable*, и набором ограничений таблицы. Вместе эти два набора составляют *схему* таблицы. В *DataTable* также входит набор *Data Rows*, где, собственно, и хранятся данные объекта *DataSet*.

Кроме того, объект *DataSet* содержит набор *Data Relations*, который позволяет создавать связи между строками различных таблиц. *DataRelations* содержит набор объектов *DataRelation*, определяющих связи между таблицами, которые составляют *DataSet*. Рассмотрим, например, *DataSet*, содержащий две связанные между собой таблицы — Customers и Orders. В таблице Customers каждый покупатель представлен только одной строкой, идентифицируемой уникальным полем CustomerlD. В таблице Orders содержатся сведения о покупателях, разместивших заказы; заказчики идентифицируются значением поля CustomerlD. Если некоторый покупатель разместит несколько заказов, то в таблице Orders может оказаться несколько записей с одинаковым значением CustomerlD. Подобная ситуация — пример связи типа «один ко многим», такую связь можно определить посредством объекта *DataRelation*.

Наконец, *DataSet* содержит набор *Extended Properties*, в котором хранятся дополнительные данные объекта *DataSet*.

Провайдер данных

Связь с БД создается и поддерживается при помощи *провайдера данных* (Data Provider). В действительности провайдер — это набор взаимосвязанных компонентов, обеспечивающих эффективный высокопроизводительный доступ к данным. В настоящее время с .NET Framework поставляются два провайдера данных: *SQL Server .NET Data Provider*, созданный для работы с SQL Server версии 7.0 и выше, и *OleDb .NET Data Provider* — для подключения к БД других типов. Любой провайдер данных состоит из близких версий следующих универсальных классов компонентов:

- *Connection* — обеспечивает подключение к БД;
- *Command* — применяется для управления источником данных; позволяет исполнять команды, не возвращающие данных, например INSERT, UPDATE и DELETE, либо команды, возвращающие объект *DataReader* (такие, как SELECT);

- *DataReader* — предоставляет доступный только для однонаправленного чтения набор записей, подключенный к источнику данных;
- *DataAdapter* — заполняет отсоединенный объект *DataSet* или *DataTable* и обновляет его содержимое.

Примечание Если в тексте этой главы название класса компонента приводится без префикса, указывающего его версию, сказанное в равной мере относится к обоим версиям провайдеров данных. Например, *Command* означает как *OleDbCommand*, так и *SqlCommand*.

Доступ к данным в ADO.NET осуществляется так: объект *Connection* устанавливает между приложением и БД соединение, напрямую доступное объектам *Command* и *DataAdapter*. Объект *Command* позволяет исполнять команды непосредственно над БД. Если исполненная команда возвращает несколько значений, *Command* открывает к ним доступ через объект *DataReader*. Полученные результаты можно обрабатывать напрямую, используя код приложения, либо через объект *DataSet*, заполнив его при помощи объекта *DataAdapter*. Для обновления БД также применяют объекты *Command* или *DataAdapter*.

В следующих разделах приводятся общие сведения об универсальных классах компонентов провайдеров данных.

Объект Connection

Объект *Connection* представляет соединение с БД. Visual Studio .NET поддерживает два класса *Connection*: *SqlConnection*, предназначенный для подключения к SQL Server версии 7.0 и выше, и *OleDbConnection*, который применяется для подключения к самым разным БД. Все данные, необходимые для открытия канала связи с БД, хранятся в свойстве *ConnectionString* объекта *Connection*, этот объект также поддерживает ряд методов, позволяющих обрабатывать данные с применением транзакций.

Объект Command

Объект *Command* также представлен двумя классами — *SqlCommand* и *OleDbCommand*. Он позволяет исполнять команды над БД, используя для обмена данными установленное соединение. При помощи объектов *Command* можно исполнять хранимые процедуры, команды SQL, а также операторы, возвращающие целые таблицы. Объект *Command* поддерживает три метода:

- *ExecuteNonQuery*. Исполняет команды, не возвращающие данные, например INSERT, UPDATE и DELETE;
- *ExecuteScalar*. Исполняет запросы к БД, возвращающие единственное значение;
- *ExecuteReader*. Возвращает результирующий набор через объект *DataReader*.

Объект DataReader

Объект *DataReader* предоставляет поток с набором записей БД, доступный только для однонаправленного чтения. Б отличие от других компонентов провайдера данных, создавать экземпляры *DataReader* напрямую не разрешается, его можно получить при помощи методов *ExecuteReader* объекта *Command*: метод *SqlCommand.ExecuteReader* возвращает объект *SqlDataReader*, *а* метод *OleDbCommand.ExecuteReader* — объект *OleDbDataReader*. Если записывать содержимое объекта *DataReader* на диск

не требуется, строки с данными можно передавать приложению напрямую Поскольку в любой момент времени в памяти находится только одна строка, использование объекта *DataReader* почти не снижает производительность системы, но требует монопольного доступа к открытому объекту *Connection* в течение времени жизни объекта *DataReader.*

Объект DataAdapter

DataAdapter — это основной класс ADO.NET, обеспечивающий доступ к отсоединенным данным. В сущности, он выполняет функции посредника во взаимодействии между БД и объектом *DataSet.* При вызове метода *Fill* объект *DataAdapter заполняет Data Table* или *DataSet* данными, полученными из БД. После обработки данных, загруженных в память, можно записать модифицированные записи в БД, вызвав метод *Update* объекта *DataAdapter.* У *DataAdapter* имеются четыре свойства, представляюших команды БД:

- *SelectCommand.* Содержит текст или объект команды, осуществляющей выборку данных из БД; при вызове метода *Fill* эта команда исполняется и заполняет объект *DataTable* или объект *DataSet;*
- *Insert Command.* Содержит текст или объект команды, осуществляющий вставку строк в таблицу;
- *DeleteCommand.* Содержит текст или объект команды, осуществляющий удаление строки из таблицы;
- *UpdateCommand.* Содержит текст или объект команды, осуществляющий обновление значений в БД.

При вызове метода *Update* все измененные данные копируются из объекта *DataSet* в БД с исполнением соответствующей команды *InsertCommand, DeleteCommand* или *UpdateCommand.*

Резюме

- ADO.NET — это эффективная и масштабируемая технология доступа к данным (главным образом, отсоединенным).
- Доступ к данным предоставляются через объект *DataSet,* представляющий отсоединенную копию БД или ее фрагмента, загруженную в памяти.
- Провайдер данных — это набор классов, обеспечивающих доступ к БД. Основные классы провайдеров данных:
  - *Connection*
  - *Command*
  - *DataReader*
  - *DataAdapter*
- В Visual Studio .NET имеются два провайдера данных:
  - SQL — содержит классы, оптимизированные для доступа к БД SQL Server версии 7.0 и выше;
  - OleDb — содержит классы, обеспечивающие доступ к разнообразным типам БД.

Занятие 2. Доступ к данным

В Visual Studio .NET имеется множество встроенных мастеров и дизайнеров, которые помогут быстро и эффективно создать архитектуру доступа к данным во время разработки и оснастить приложение надежным механизмом доступа к данным, затратив минимум усилий на написание кода. Наряду с этим все возможности объектной модели ADO.NET доступны программно, что позволяет реализовать нестандартные функции или создавать приложения, ориентированные на нужды пользователя. На этом занятии вы узнаете, как с помощью ADO.NET подключиться к БД, извлечь из нее данные и передать их приложению. Вы научитесь делать это как с помощью инструментов с графическим интерфейсом из Visual Studio .NET, так и с использованием прямого программного доступа.

Изучив материал этого занятия, вы сможете:

- ✓ установить соединение с БД;
- ✓ создать и настроить объект *DataAdapter*;
- ✓ создать объект *Command,* использующий команды SQL;
- ✓ создать объект *Command,* вызывающий хранимые процедуры из БД;
- ✓ создать типизированный объект *DataSet*;
- ✓ заполнить объект *DataSet* с помощью объекта *DataAdapter*;
- ✓ реализовать программный доступ к данным через объект *DataReader.*

Продолжительность занятия - около 60 мин.

Подключение к БД

Есть несколько способов подключения к БД в приложении. Проще всего это сделать при помощи графических инструментов Visual Studio .NET во время разработки.

Для управления текущими соединениями с источниками данных служит окно Server Explorer. Оно обычно расположено у левой границы окна IDE и соседствует с панелью Toolbox. Если в данный момент в окне IDE отображается панель Toolbox, щелкните расположенный под ней ярлычок Server Explorer, чтобы открыть одноименное окно.

Окно Server Explorer (без стыковки) показано на рис. 6-1.

Текущие соединения с источниками данных, доступные в Visual Studio .NET, отображаются в окне Server Explorer в виде узлов дерева Data Connections. Чтобы добавить к проекту соединение, достаточно перетащить нужный узел Data Connections из окна Server Explorer в окно дизайнера формы. При этом автоматически создается новый объект *SqlConnection* или *OleDbConnection*, который конфигурируется для подключения к нужной БД.

► **Как добавить соединение при помощи окна Server Explorer**

Перетащите узел, представляющий нужную БД, в окно дизайнера.

Окно Server Explorer также позволяет создать новое соединение, щелкнув правой кнопкой узел Data Connections и выбрав из контекстного меню команду Add Connection. При этом открывается диалоговое окно Data Link Properties, показанное на рис, 6-2.

Окно Data Link Properties предоставляет графический интерфейс для настройки соединений с источником данных. На вкладке Providers можно выбрать провайдер

для подключения к БД, на вкладке Connection — настроить свойства соединения, а на вкладке Advanced — задать дополнительные необязательные свойства. Вкладка All позволяет просматривать и редактировать текущие параметры для всех свойств одновременно. Настроив новое подключение, щелкните OK, чтобы закрыть диалоговое окно Data Link Properties, — новое соединение появится в окне Server Explorer, откуда его можно перетащить в окно дизайнера.

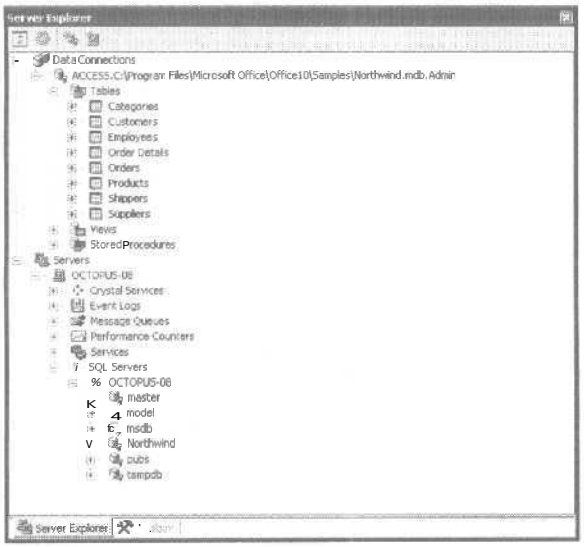


Рис. 6-1. Server Explorer

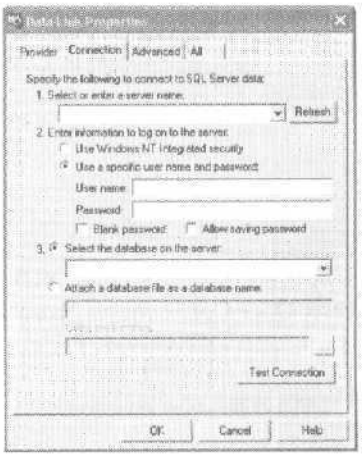


Рис. 6-2. Диалоговое окно Data Link Properties

▶ Создание нового соединения

1. В окне Server Explorer щелкните правой кнопкой узел Data Connections и выберите из контекстного меню команду Add Connection — откроется окно Data Link Properties.
2. На вкладке Provider выберите провайдер соответствующего типа.

3. На вкладке Connection выберите БД, к которой вы хотите подключиться. В зависимости от типа провайдера иногда требуется установить дополнительные свойства, например ввести имя сервера или пароль.
4. Выставив все параметры, щелкните кнопку Test Connection, чтобы проверить работу соединения.
5. Перетащите новое соединение из окна Server Explorer в окно дизайнера — будет создан и настроен объект соответствующего типа, представляющий это соединение.

Соединение с БД также допустимо создавать вручную. Для этого перетащите объект *Connection* с панели Toolbox в окно дизайнера либо объявите и создайте экземпляр этого объекта в коде. В этом случае вам придется вручную устанавливать свойство *ConnectionString*. Ниже показан пример, демонстрирующий программное создание объекта *Connection* и установку свойства *ConnectionString*:

Visual Basic .NET

```
' Объявить класс OleDbConnection и создать его экземпляр.
Dim myConnection As New OleDbConnection()
' Создать строку подключения, в которой задан тип БД.
' БД Microsoft Access и указан путь к файлу БД.
myConnection.ConnectionString = _
    "Provider=Microsoft.Jet.OLEDB.4.0;" & _
    "DataSource=C:\Northwind\Northwind.mdb"
```

Visual C#

```
// Объявить класс OleDbConnection и создать его экземпляр.
OleDbConnection myConnection = new 01eDbConnection();
// Создать строку подключения, в которой задан тип БД.
// БД Microsoft Access и указан путь к файлу БД.
myConnection.ConnectionString =
    "Provider=Microsoft.Jet.OLEDB.4.0;DataSource=" +
    "C:\\Northwind\\Northwind.mdb";
```

Примечание Набор параметров в строке подключения может меняться в зависимости от типа и конфигурации провайдера данных, поэтому рекомендуется по возможности создавать соединения с помощью графических инструментов Visual Studio .NET.

▶ **Как создать новое соединение программно**

1. Объявите и создайте экземпляр объекта *Connection* соответствующего типа.
2. Установите значение свойства *ConnectionString*.

▶ **Как создать новое подключение с помощью графического интерфейса дизайнера**

1. Перетащите объект *Connection* соответствующего типа с вкладки Data панели Toolbox в окно дизайнера.
2. В окне Properties установите свойство *ConnectionString*.

Команды для манипулирования данными

Объект *Command* содержит ссылку на хранимую процедуру БД или оператор SQL и способен исполнить этот оператор на источнике данных, используя активное соединение. Объект *Command также* содержит все данные, необходимые для исполнения команды: ссылку на активное соединение, текст команды и ее параметры.

Подобно другим классам компонентов провайдера данных, класс *Command* представлен двумя вариантами. Первый, *OleDbCommand*, разработан для взаимодействия с самыми разными типами БД, а *SqlCommand* — только для БД SQL Server 7.0 и выше.

Для класса *Command* достаточно активного соединения, взаимодействие с объектом *DataAdapter* ему не требуется. В силу этих причин объекты *Command* очень быстро и эффективно взаимодействуют с различными БД, позволяя:

- исполнять команды, не возвращающие значения, например INSERT, UPDATE и DELETE;
- исполнять команды, возвращающие единственное значение;
- исполнять команды языка Database Definition Language (DDL), например CREATE TABLE и ALTER;
- работать с объектом *DataAdapter*, возвращающим объект *DataSet*;
- возвращать результирующий набор непосредственно через экземпляр объекта *DataReader* — это самый быстрый способ доступа к данным, он особенно удобен, если данные требуются только для чтения;
- возвращать результирующий набор в виде потока XML — эту возможность поддерживает только класс *SqlCommand;*
- возвращать результирующий набор, созданный на основе нескольких таблиц или в результате исполнения нескольких операторов.

Создание и настройка команд

Есть три способа создания команды для манипулирования данными:

- перетаскивание хранимой процедуры из окна Server Explorer в окно дизайнера;
- перетаскивание объекта *SqlCommand (или OleDbCommand)* с вкладки Data панели Toolbox в окно дизайнера с последующей его настройкой;
- объявление и создание экземпляра объекта *Command* соответствующего типа в коде с последующей его настройкой вручную.

Первый способ — самый простой. Он позволяет автоматически создавать объект *Command* на основе любой хранимой процедуры БД. В результате перетаскивания хранимой процедуры в окно дизайнера создается объект *Command* соответствующего типа. Новый объект команды ссылается на хранимую процедуру, позволяя вызывать ее в любое время без дополнительной настройки.

Чтобы создать объект *Command,* ссылающийся на имеющуюся хранимую процедуру, перетащите хранимую процедуру из окна Server Explorer в окно дизайнера — в результате будет создан и сконфигурирован экземпляр объекта *Command.*

Создание объекта *Command* нужного типа при помощи интерфейса дизайнера не многим сложнее: перетащите *SqlCommand* или *OleDbCommand* с вкладки Data панели Toolbox в окно конструктора. Создав объект *Command,* сконфигурируйте его, установив свойства *Connection, CommandType* и *CommandText.*

Свойство *CommandType* определяет тип команды, содержащийся в свойстве *CommandText,* оно может принимать одно из следующих значений:

- **Text** — заставляет рассматривать значение свойства *CommandText* как текст команды SQL, при этом в свойстве *CommandText* должен быть один или несколько допустимых операторов SQL, разделенных точкой с запятой. В последнем случае операторы SQL выполняются по порядку;
- **StoredProcedure** — в этом случае в свойстве *CommandText* необходимо указать имя существующей хранимой процедуры — она будет исполнена при вызове данной команды;
- **TableDirect** — в этом случае в свойстве *CommandText* должно быть имя одной или нескольких таблиц. При исполнении эта команда вернет все столбцы и строки таблиц, заданных свойством *CommandText*.

Свойство *Connection* следует устанавливать в соответствии с типом активного соединения: для объекта *SqlCommand* требуется соединение на основе объекта *SqlConnection*, а для *OleDbCommand* — соединение на основе *OleDbConnection*.

Исполнение команд

Оба типа объектов *Command* поддерживают три метода, позволяющих исполнить команду, которую он представляет:

- *ExecuteNonQuery*
- *ExecuteScalar*
- *ExecuteReader*

Класс *SqlCommand* дополнительно поддерживает метод *ExecuteXmlReader*.

Все эти методы исполняют на источнике данных команду, представленную объектом *Command*, отличаются они возвращаемым значением. Метод *ExecuteNonQuery* — самый простой из них, он не возвращает никаких значений. Этот метод обычно применяют для вызова команд SQL и хранимых процедур, таких, как INSERT, UPDATE или DELETE. Кроме того, вызов этого метода — единственный способ исполнения в ADO.NET команд DDL, например CREATE или ALTER. Метод *ExecuteScalar* возвращает только значение первого поля первой строки, извлеченной заданной командой, независимо от того, сколько строк выбрано этой командой в действительности. *ExecuteReader* возвращает неизменяемый объект *DataReader*, допускающий только последовательный однонаправленный просмотр без использования объекта *DataAdapter*. Если не требуется модифицировать содержимое БД или как-то иначе манипулировать им, этот способ извлечения данных является самым быстрым и эффективным. Класс *SqlCommand* поддерживает дополнительный метод — *ExecuteXmlReader*, возвращающий результирующий набор в формате XML; результаты возвращаются в виде неизменяемого объекта *XmlReader*, доступного только для последовательно однонаправленного просмотра.

Параметры

Команды для манипулирования данными часто вызываются с параметрами, при этом некоторые элементы команды становятся известными только в период выполнения, Давайте рассмотрим приложение для учета товаров в книжном магазине. В нем следует предусмотреть функцию поиска книг по названию, которую можно реализовать посредством запроса к БД на основе подобного оператора SQL:

```
SELECT * FROM Books WHERE (Title LIKE [value])
```

Уже в период разработки ясно, что потребуется возможность поиска книг по названию, заданному пользователем в период выполнения. Поскольку значение, ко-

торое введет пользователь, заранее неизвестно, необходим некоторый механизм для передачи введенного значения оператору SQL во время выполнения.

Параметры — это значения, которыми заполняют поля подстановки, введенные в текст команды во время разработки. Каждый параметр представлен экземпляром класса *OleDbParameter* или *SqlParameter*, в зависимости от типа провайдера. Параметры хранятся в свойстве *Parameters* объекта *Command*. В период выполнения значения параметров считываются из этого свойства и подставляются в оператор SQL либо передаются хранимой процедуре.

Объекты *Command* поддерживают набор Parameters, в котором хранятся объекты *Parameter* соответствующего типа. Ниже перечислены некоторые свойства объектов *Parameter*.

- *DbType* (не отображается в окне дизайнера)
- *Direction*
- *OleDb Type* (только для объекта *OleDb Parameters)*
- *ParameterName*
- *Precision*
- *Scale*
- *Size*
- *SourceColumn*
- *Source Version*
- *SQL Type* (только для *SQLParameters)*
- *Value*

Свойства *DbType* и *OleDb Type* объекта *OleDb Parameters* взаимосвязаны. Первое представляет тип параметра так, как тот представлен в общей системе типов (CTS), а второе — как он представлен в БД; это необходимо, поскольку не все БД совместимы с CTS. Объект *Parameter* выполняет преобразование параметров из типа, используемого в приложении, в тип, используемый в БД. Поскольку эти свойства взаимосвязаны, при изменении значения одного из них значение другого автоматически изменяется и преобразуется в соответствующий поддерживаемый тип. Аналогичным образом связаны свойства *DbType* и *SqlType* объектов *SqlParameter*; свойство *SqlType* указывает тип БД SQL, представленный этим параметром.

Свойство *Direction* объекта параметра определяет, является ли этот параметр входным или выходным. Возможные значения для этого свойства — *Input, Output, InputOutput* или *ReturnValue* — указывают, должен ли параметр хранить значение, возвращаемое хранимой процедурой или функцией.

На члены набора Parameters ссылаются в коде по индексу либо по имени, заданному свойством *ParameterName*. Ниже показаны два способа установки значения первого по счету параметра с именем *myParameter*.

Visual Basic .NET

```
' Эта команда устанавливает значение параметра, ссылаясь на него по индексу.
OleDbCommand1.Parameters(0).Value = "Hello World"
' Эта строка устанавливает значение параметра, ссылаясь на него по имени.
OleDbCommand1.Parameters("myParameter").Value = "Goodbye for now"
```

Visual C#

```
// Эта команда устанавливает значение параметра, ссылаясь на него по индексу.
OleDbCommand1.Parameters[0].Value = "Hello World";
```

```
// Эта строка устанавливает значение параметра, ссылаясь на него по имени,
OleDbCommand1.Parameters["myParameter"].Value = "Goodbye for now";
```

Свойства *Precision, Scale* и *Size* определяют длину и точность значения параметров. *Precision* и *Scale* применяются с числовыми и десятичными параметрами и определяют разрядность и длину дробной части значения свойства *Value* соответственно, а *Size* применяется с двоичными и строковыми параметрами и представляет максимальную длину такого поля.

Свойства *SourceColumn и SourceVersion* используются с параметрами, привязанными к полям объекта *DataTable. SourceColumn* указывает поле для поиска или сопоставления значений, а *SourceVersion* — версию поля для редактирования.

Свойство *Value* содержит значение параметра.

Если свойство *CommandType* объекта *Command* установлено в *Text,* необходимо предусмотреть поля подстановки для всех параметров оператора SQL. В случае объекта *OleDbCommand* поле подстановки обозначается символом «?», например:

```
SELECT EmpId, Title, FirstName, LastName
FROM Employees
WHERE (Title = ?)
```

Здесь знаком вопроса указано место подстановки параметра. Допускается также указывать несколько параметров следующим образом:

```
SELECT EmpId, Title, FirstName, LastName
FROM Employees
WHERE (FirstName = ?) AND (LastName = ?)
```

Если в тексте команды имеется несколько полей для подстановки параметров, порядок их заполнения определяется порядком элементов набора *Parameters.*

С объектом *SqlCommand* применяют именованные параметры. Чтобы создать поле подстановки для именованного параметра, необходимо указать имя параметра (так. как оно задано свойством *ParameterName*), предварив его символом «@». Например, в следующем операторе SQL объявлено поле для подстановки именованного параметра Title:

```
SELECT EmpId. Title, FirstName, LastName
FROM Employees
WHERE (Title = @Title)
```

Ниже описаны способы исполнения команд различных типов при помощи объектов *Command.*

▶ **Как при помощи объекта *Command* исполнить команду, не возвращающую значений**

Так исполняют команды INSERT, UPDATE и DELETE, а также команды DDL, такие, как CREATE TABLE и ALTER.

1. Для исполнения хранимой процедуры установите свойство *CommandType* в *Stored Procedure*, *а* для исполнения оператора SQL — в *Text.*
2. Запишите в свойство *CommandText* имя хранимой процедуры или строку оператора SQL соответственно.
3. Объявите все необходимые параметры и присвойте им соответствующие значения.
4. Вызовите метод *Command.ExecuteNonQuery*, например, так:

Visual Basic .NET

```
' Эту команду одинаково вызывают при использовании
' как класса OleDbCommand, так и SqlCommand.
myCommand.ExecuteNonQuery()
```

Visual C#

```
// Эту команду одинаково вызывают при использовании
// как класса OleDbCommand, так и SqlCommand.
myCommand.ExecuteNonQuery();
```

▶ **Как при помощи объекта *Command* исполнить** команду, **возвращающую единственное значение**

1. Для исполнения хранимой процедуры установите свойство *CommandType* в *StoredProcedure*, а для исполнения оператора SQL — в *Text*.
2. Запишите в свойство *CommandText* имя хранимой процедуры или строку оператора SQL соответственно.
3. Объявите все необходимые параметры и присвойте им соответствующие значения.
4. Вызовите метод *Command.ExecuteScalar*, например, так:

Visual Basic .NET

```
' Эту команду одинаково вызывают при использовании
' как класса OleDbCommand, так и SqlCommand.
Dim O As Object
O = myCommand.ExecuteScalar()
```

Visual C#

```
// Эту команду одинаково вызывают при использовании
// как класса OleDbCommand, так и SqlCommand.
Object O;
O = myCommand.ExecuteNonScalar();
```

Применение объекта DataReader

Исполнить с помощью объекта *Command* команду, не возвращающую значений или возвращающую единственное значение, довольно просто. Чтобы использовать объект *Command* с запросами, возвращающими несколько значений, обратитесь к методу *Execute Reader*. Он возвращает объект *DataReader*.

DataReader — это упрощенный объект, обеспечивающий быстрое и эффективное последовательное однонаправленное чтение данных. Объект *DataReader* позволяет перебирать записи результирующего набора, передавая нужные значения напрямую коду приложения. При этом *DataSet* намного эффективнее, но менее гибок по сравнению с *DataAdapter*. Данные, полученные через объект *DataSet*, доступны только для чтения, поэтому их нельзя модифицировать средствами *DataSet*. При этом данные разрешается просматривать только в одном направлении: нельзя вернуться к записи, прочитанной ранее. Кроме того, объект *DataReader* ориентирован на использование постоянных соединений и, пока он существует, требует монопольного доступа к активному соединению.

Создание объекта DataReader

Объект *DataReader* нельзя создать напрямую, это делается путем вызова метода *ExecuteReader* объекта *Command*. Подобно другим членам классов провайдеров данных, у каждого класса *DataProvider* есть собственный класс *DataReader*. объект *OleDbCommand* возвращает *OleDbDataReader*, а объект *SqlCommand* — *SqlDataReader*, например:

Visual Basic .NET

```
' Этот пример предполагает наличие объектов OleDbCommand и
' SqlCommand с именами myOleDbCommand and mySqlCommand соответственно.
Dim myOleDbReader As System.Data.OleDb.OleDbDataReader
Dim mySqlReader As System.Data.SqlClient.SqlDataReader
' Эта команда создает новый объект OleDbReader и записывает его в переменную.
myOleDbReader = myOleDbCommand.ExecuteReader()
' Эта команда создает новый объект SqlReader и записывает его в переменную.
mySqlReader - mySqlCommand.ExecuteReader()
```

Visual C#

```
// Этот пример предполагает наличие объектов OleDbCommand и
// SqlCommand с именами myOleDbCommand and mySqlCommand соответстаенно.
System.Data.OleDb.OleDbDataReader inyOleDbReader;
System.Data.SqlClient.SqlDataReader mySqlReader;
// Эта команда создает новый объект OleDbReader и записывает его в переменную.
myOleDbReader - myOleDbCommand.ExecuteReader();
// Эта команда создает новый объект SqlReader и записывает его в переменную.
mySqlReader = mySqlCommand.ExecuteReader();
```

При вызове метода *ExecuteReader* объект *Command* исполняет представленную им команду и создает объект *DataReader* соответствующего типа, который можно записать в переменную ссылочного типа.

Организация простого доступа к данным с помощью объекта DataReader

Получив ссылку на объект *DataReader*, можно просматривать записи, загружая нужные данные в память. У нового объекта *DataReader* указатель чтения устанавливается на первую запись результирующего набора. Чтобы сделать ее доступной, следует вызвать метод *Read*. Если запись доступна, метод *Read* переводит указатель объекта *DataReader* к следующей записи и возвращает *True (true)*, в противном случае метод *Read* возвращает *False (false)*. Таким образом, метод *Read* используют для перебора записей в цикле *While (while)*, например, так:

Visual Basic .NET

```
While myDataReader.Read()
    ' Этот код будет однократно исполнен
    ' для каждой записи результирующего набора.
End While
```

Visual C#

```
while (myDataReader.Read())
{
   // Этот код будет однократно исполнен
   // для каждой записи результирующего набора.
}
```

При чтении записи с помощью объекта *DataReader* значения отдельных полей доступны через индексатор или свойство по умолчанию в виде массива объектов, к элементам которого разрешается обращаться по индексу либо по имени поля, например:

Visual Basic .NET

```
While myDataReader.Read()
   Dim myObject As Object = myDataReader(3)
   Dim myOtherObject As Object - myDataReader("CustomerID")
End While
```

Visual C#

```
while (myDataReader.Read())
{
   object myObject = myDataReader[3];
   object myOtherObject - myDataReader["CustomerID"];
}
```

При этом способе доступа *DataReader* предоставляет все значения в виде объектов, хотя из *DataReader* допустимо извлекать и типизированные данные. Более подробно об этом рассказано далее.

Прочитав данные с помощью *DataReader,* следует вызвать метод *Close,* чтобы закрыть *DataReader*; в противном случае объект *DataReader* будет удерживать монопольный доступ к активному соединению, сделав его недоступным другим объектам. При вызове метода *Execute Reader,* установив свойство *Command Behavior* в *CloseConnection*, вы автоматически закроете соединение, не вызывая метод *Close* явно:

Visual Basic .NET

```
myDataReader.Close()
```

Visual C#

```
myDataReader.Close();
```

Выборка столбца с помощью объекта DataReader

Следующий пример демонстрирует перебор записей результирующего набора для вывода содержимого одного из столбцов в окне Console. Этот пример предполагает наличие объекта *OleDbCommand* с именем *myOleDbCommand*, свойство *Connection* которого определяет соединение с именем *myConnection.*

Visual Basic .NET

```
' Открыть активное соединение.
myConnection.Open()
' Создать объект DataReader и записать его в переменную myReader.
Dim myReader As System.Data.OleDb.OleDbDataReader - _
   myOleDbCommand.ExecuteReader()
' Вызвать метод Read перед попыткой чтения данных.
While myReader.Read()
   ' Получить доступ к полю можно по имени
   ' либо по его порядковому номеру.
   Console.WriteLine(myReader("Customers").ToString())
End While
' Обязательно закрывайте DataReader, закончив работу с ним,
myReader.Close()
' и соединение, если оно больше не нужно.
myConnection.Close
```

Visual C#

```
// Открыть активное соединение,
myConnection.Open();
// Создать объект DataReader и записать его в переменную myReader.
System.Data.OleDb.OleDbDataReader myReader =
   myOleDbCommand.ExecuteReader();
// Вызвать метод Read перед попыткой чтения данных.
while (myReader.Read())
{
   // Получить доступ к полю можно по имени
   // либо по его порядковому номеру.
   Console.WriteLine(myReader["Customers"].ToString());
}
// Обязательно закрывайте DataReader, закончив работу с ним,
myReader.Close();
// и соединение, если оно больше не нужно.
myConnection.Close;
```

▶ **Как получить доступ к данным с помощью объекта *DataReader***

1. Вызовите метод *Execute Reader* объекта *Command* и запишите результирующий объект *DataReader* в переменную соответствующего типа.
2. Обработайте результирующий набор в цикле *While* (*while*), выполняя любые действия над данными внутри цикла. Прежде чем обращаться к данным, следует вызвать метод *Read* объекта *DataReader.*
3. Закончив, вызовите метод *Close* объекта *DataReader*, чтобы закрыть соединение.

Извлечение типизированных данных с помощью объекта DataReader

DataReader не только предоставляет данные в виде объектов, но и поддерживает и собственные методы для извлечения данных из результирующего набора. Имена

этих методов образуются из префикса *Get* и имени типа извлекаемых данных. Например, метод, извлекающий значения типа *Boolean,* назван *GetBoolean.* Если известен тип данных некоторого столбца, можете воспользоваться методами объекта *DataReader* для извлечения из этого столбца строго типизированных данных. Например:

Visual Basic .NET

```
Dim myString As String
myString = myDataReader.GetBoolean(3)
```

Visual C#

```
string myString;
myString = myDataReader.GetBoolean(3);
```

При использовании такого способа извлечения данных вы должны указать порядковый номер, а не имя поля. Если известно только имя поля, можно определить его порядковый номер, вызвав метод *GetOrdinal,* например, так:

Visual Basic .NET

```
Dim CustomerID As Integer
Dim Customer As String
' Определить порядковый номер поля 'CustomerID'.
CustomerID = myDataReader.GetOrdinal("CustomerID")
' Извлечь строку из этого поля и записать ее в переменную Customer.
Customer = myDataReader.GetString(CustomerID)
```

Visual C#

```
int CustomerID;
string Customer;
// Определить порядковый номер поля 'CustomerID'.
CustomerID = myDataReader.GetOrdinal("CustomerID");
// Извлечь строку из этого поля и записать ее в переменную Customer.
Customer = myDataReader.GetString(CustomerID);
```

▶ **Как извлечь типизированные данные с помощью объекта *DataReader***

1. При необходимости определите порядковый номер поля, вызвав метод *GetOrdinal* и передав ему имя нужного поля.
2. Вызовите соответствующий метод *Get* объекта *DataReader,* указав порядковый номер поля, значение которого хотите вернуть.

Работа с несколькими результирующими наборами

Если свойство *CommandType* объекта *Command* установлено в *Text,* вернуть несколько результирующих наборов можно единственной командой. Для этого нужно поместить в свойство *CommandText* несколько команд SQL, разделив их точкой с запятой, например:

```
SELECT *
FROM Accounts;
```

```
SELECT *
FROM Creditors
```

Если свойство *CommandText* объекта *Command* содержит несколько операторов SQL, эти операторы исполняются по порядку. Если результирующий набор возвращают несколько операторов, то объект *DataReader* также вернет несколько результирующих наборов в том порядке, в котором были исполнены операторы.

Первый результирующий набор объект *DataReader* возвращает автоматически. Чтобы получить доступ к следующим результирующим наборам, необходимо вызывать метод *NextResult.* Как и *Read,* метод *NextResult* возвращает *False* (*false*)при попытке обращения к несуществующему результирующему набору, но, в отличие от *Read,* он устанавливает указатель объекта *DataReader* на первый результирующий набор. Если перед чтением первого результирующего набора вызвать метод *NextResult*, первый набор пропускается, вместо него прочитан будет следующий результирующий набор. Ниже показан перебор результирующих наборов с помощью метода *NextResult:*

Visual Basic .NET

```
Do
    While myReader.Read()
        ' Здесь должен быть код для перебора записей текущего результирующего набора.
    End While
    ' Эта команда переходит к следующему набору результатов или возвращает False
    ' при попытке перехода к несуществующему результирующему набору.
Loop While myReader.NextResult()
```

Visual C#

```
do
{
    while (myReader.Read())
    {
        // Здесь должен быть код для перебора записей текущего результирующего
набора.
    }
} while (myReader.NextResult());
```

▶ **Как прочитать несколько результирующих наборов с помощью объекта *DataReader***

1. Установите значение свойства *CommandType*объекта *Command* в Text.
2. Поместите в свойство *CommandText* объекта *Command* несколько операторов SQL, разделяя их точкой с запятой.
3. Вызовите метод *ExecuteReader* объекта *Command* и запишите полученный *DataReader* в переменную.
4. Выполните перебор результирующих наборов при помощи метода *DataReader* объекта *NextResult.*

Формирование запросов SQL во время выполнения

Иногда конструкция запроса SQL становится известной только в период выполнения. Например, запрос может использовать строку для поиска, введенную пользователем, или возвращать столбцы и таблицы, определенные программно в период выполнения. Для решения этих задач соответствующие команды создают, настраивают и исполняют в период выполнения.

Первый шаг — формирование строки команды. Напишите сначала заготовку команды; при необходимости заменяйте значения, которые должны быть вставлены в период выполнения, строковыми переменными; для объединения строк используйте оператор конкатенации. Вот пример:

Visual Basic .NET

```
Dim Cmd As String
Cmd = "SELECT * FROM Employees WHERE Name = '" & aString &.....
```

Visual C#

```
string Crnd;
Cmd = "SELECT * FROM Employees WHERE Name = '" + aString +....;
```

Внимание! Любые строковые переменные, передаваемые базе данных при помощи конструкции WHERE, необходимо заключать в одинарные кавычки ('). Если одинарные кавычки находятся внутри такой переменной, следует заменить их парой одинарных кавычек (''), в противном случае исполнение запроса закончится неудачей,

У каждого провайдера данных есть конструктор класса *Command*, позволяющий устанавливать свойства *CommandText* и *Connection* при создании экземпляра *Command*. После установки этих свойств вам остается лишь открыть соединение и исполнить команду. Следующий пример демонстрирует получение методом строки поиска и применение полученной строки для создания и исполнения команды SQL:

Visual Basic .NET

```
' Этот пример предполагает наличие соединения myConnection и
' оператора, импортирующего пространство имен System.Data.OleDb.
Public Sub DeleteRecord(aString As String)
   Dim Cmd As String
   Cmd = "DELETE * FROM Employees WHERE Name = '" & aString &.....
   ' Теперь нужно передать переменную Cmd как строку команды
   ' и myConnection как объект соединения;
   Dim myCommand As New OleDbCommand(Cmd, myConnection)
   ' осталось открыть соединение и исполнить команду.
   myConnection.Open()
   myCommand.ExecuteNonQuery()
   ' Не забудьте закрыть соединение!
   myConnection.Close()
End Sub
```

Visual C#

```
// Этот пример предполагает наличие соединения myConnection и
// оператора, импортирующего пространство имен System.Data.OleDb.
public void DeleteRecord(string aString)
{
   string Cmd;
   Cmd = "DELETE * FROM Employees WHERE Name = '" + aString + ..... ;
   // Указывает Cmd в качестве командной строки, и myConnection
   // в качестве подключения
   OleDbCommand myCommand = new OleDbCommand(Cmd, myConnection);
   II осталось открыть соединение и исполнить команду,
   myConnection.Open();
   myCommand.ExecuteNonQuery();
   /7 Не забудьте закрыть соединение!
   myConnection.Close();
}
```

Создание и настройка объекта DataAdapter

Объекты *DataAdapter* обеспечивают связь между источником данных и объектом *DataSet,* управляя обменом данных и контролируя их передачу. Одни приложения осуществляют одностороннюю передачу данных, а другим требуется не только извлекать, но и модифицировать данные источника. Объект *DataAdapter* способен извлекать данные, заполнять объекты *DataSet* и при необходимости обновлять содержимое источникаданных.

В Visual Studio .NET имеется две разновидности объекта *DataAdapters*: *SqlDataAdapter*, оптимизированный для взаимодействия с БД SQL Server 7.0 и выше, и *OleDb DataAdapter,* способный предоставить доступ к любому источнику данных, поддерживаемому *OleDb Provider.*

Как правило, обменом данными в каждой паре «объект *DataTabie* набора *DataSet* — таблица БД» управляет отдельный объект *DataAdapter.* В *DataSet* может быть несколько таблиц, поэтому для каждой, добавляемой к объекту *DataSet,* следует создавать собственный объект *DataAdapter.*

Существует три способа создания *DataAdapter.* перетаскиванием элемента БД из окна Server Explorer, при помощи нового мастера Data Adapter Configuration или ручным объявлением и настройкой этого объекта в коде. Более подробно эти способы обсуждаются в следующих разделах.

Создание DataAdapter с помощью окна Server Explorer

Проще всего создать *DataAdapter,* воспользовавшись окном Server Explorer. Узел Data Connections дерева, отображаемого в окне Server Explorer, содержит все доступные соединения с данными в виде дочерних узлов Data Connection. Раскрыв узел соединения, вы получите дополнительные сведения о подключенной БД, в том числе список доступных таблиц, представлений и хранимых процедур. Чтобы создать объект *DataAdapter,* представляющий таблицу, просто перетащите таблицу из окна Server Explorer в окно дизайнера. Вы создадите объект *DataAdapter* соответствующего типа (*SqlDataAdapter* или *OleDb DataAdapter)* с корректными значениями свойств *SelectCommand*, *UpdateCommand, InsertCommand* и *DeleteCommand.*

Примечание Объект *DataAdapter* также можно сгенерировать на основе представления, однако этот способ не рекомендуется для получения доступа к данным для чтения и записи, поскольку дизайнер часто испытывает затруднения при генерации команд UPDATE, INSERT и DELETE для нескольких таблиц.

Кроме того, предусмотрена настройка объекта *DataAdapter,* чтобы он возвращал подмножество столбцов таблицы. Для этого раскройте узел, представляющий таблицу, и выберите нужные столбцы, щелкая соответствующие элементы с нажатой клавишей Ctrl. Затем перетащите выбранные столбцы в окно дизайнера — вы получите объект *DataAdapter,* сконфигурированный для доступа к выбранным столбцам.

▶ **Создание объекта *DataAdapter* с помощью окна Server Explorer**

1. В окне Server Explorer раскройте узел, представляющий соединение с БД, которую использует ваше приложение.
2. Раскройте узел Tables, чтобы отобразить таблицы этой БД.
3. Выберите нужную таблицу. Если вся таблица не требуется, раскройте ее узел, чтобы получить список столбцов. Выберите нужные столбцы, щелкая их при нажатой клавише Ctrl.
4. Перетащите выбранные столбцы в окно дизайнера, чтобы создать и настроить новый экземпляр объекта *DataAdapter.*

Создание DataAdapter с помощью мастера Data Adapter Configuration Wizard

Можно также создать новый объект *DataAdapter,* перетащив соответствующий объект *DataAdapterc* панели Toolbox в окно дизайнера, чтобы вызвать мастер Data Adapter Configuration.

▶ **Создание объекта *DataAdapter c* помощью мастера Data Adapter Configuration Wizard**

1. Перетащите объект *DataAdapter* соответствующего типа с вкладки Data панели Toolbox в окно дизайнера — запустится мастер Data Adapter Configuration. В окне мастера щелкните кнопку Next, чтобы настроить объекта *DataAdapter.*
2. На странице Choose Your Data Connection выберите в списке Data Connection источник данных и щелкните Next.

Примечание Если в этом списке не оказалось нужной вам БД, можно создать новое соединение. Для этого откройте окно Data Link Properties, щелкнув кнопку New Connection.

3. На странице Choose a Query Type выберите способ взаимодействия объекта *DataAdapter c* БД (таблица 6-1).

   В зависимости от типа провайдера данных некоторые способы оказываются недоступными — в этом случае они выделяются серым цветом. Выберите нужный способ и щелкните кнопку Next.

Таблица 6-1. Способы взаимодействия объекта *DataAdapter* с базой данных

Способ	Описание
Use SQL statements	Мастер позволяет указать операторы SELECT, определяющие данные, которые извлекаются объектом *DataAdapter*, а затем генерирует соответствующие команды INSERT, UPDATE и DELETE
Create new stored procedures	Мастер позволяет задать оператор SELECT для создания новой хранимой процедуры, после чего генерирует новые хранимые процедуры для команд INSERT, UPDATE и DELETE
Use existing stored procedures	Мастер позволяет задать существующие хранимые процедуры для команд **INSERT,** UPDATE и DELETE

4. Далее, в зависимости от указанного варианта, откроется одна из следующих панелей.
   - **Use SQL statements или Create new stored procedures:** откроется панель, генерирующая операторы SQL. Введите соответствующие операторы SELECT или щелкните кнопку Query Builder, чтобы сконструировать команду с помощью SQL Query Builder — будут сгенерированы соответствующие команды INSERT, DELETE и UPDATE или хранимые процедуры.

   Примечание SQL Query Builder — это графический конструктор операторов SQL, облегчающий их создание. Если вы не знаете языка SQL, настоятельно рекомендуем вам использовать этот инструмент.

   - **Use existing stored procedures:** откроется страница Bind Commands to Existing Stored Procedures с четырьмя раскрывающимися списками. Выберите в них соответствующие хранимые процедуры для команд SELECT, UPDATE, INSERT and DELETE.
5. Щелкните кнопку Finish — ваш объект *DataAdapter* сконфигурирован и готов к использованию.

Извлечение данных с помощью объектов DataAdapter

Хотя объект *DataSet* не соединен с их источником данных, он представляет полный набор данных, включая таблицы, связи между ними и ограничения. Поскольку *DataSet* является отсоединенным представлением, он способен хранить информацию из нескольких источников. Все взаимодействие между *DataSet* и БД протекает под контролем объекта *DataAdapter.*

Объект *DataAdapter* инкапсулирует функциональность, необходимую для заполнения *DataSet* данными и обновления БД, играя роль «моста» между соединением и объектом *DataSet,* В свойстве *Connection* объекта *DataAdapter* хранится ссылка на объект соединения, представляющий БД и позволяющий манипулировать ее содержимым. Объект *DataAdapter* также поддерживает свойство *SelectCommand.* Это свойство содержит объект *Command,* который представляет команды, осуществляющие выборку данных из источника. Как и у других компонентов провайдера данных, реализация объекта *DataAdapter* специфична для типа провайдера: *SqlDataAdapter* разработан специально для взаимодействия с SQL Server 7.0 и выше, а *OleDbDataAdapter* позволяет подключать самые разные источники данных.

Чтобы заполнить *DataSet* данными, следует вызвать метод *Fill* объекта *DataAdapter*. Этот метод исполняет команды, заданные свойством *SelectCommand*, на соединении, указанном свойством *Connection*, и заполняет *DataSet* данными, которые возвращает исполненная команда. Методу *Fill* необходимо передать целевой объект, которым может быть *DataSet* либо *DataTable*, например:

Visual Basic .NET

```
' Этот пример предполагает наличие адаптера данных
' с именем myDataAdapter.
Dim myDataSet As New DataSet()
' Исполнить команду, записанную а свойство SelectCommand,
' и заполнить результатами объект myDataSet.
myDataAdapter.Fill(myDataSet)
```

Visual C#

```
// Этот пример предполагает наличие адаптера данных
// с именем myDataAdapter.
DataSet myDataSet = new DataSet();
// Исполнить команду, записанную в свойство SelectCommand,
// и заполнить результатами объект myDataSet.
myDataAdapter.Fill(myDataSet);
```

Заметьте: взаимодействия с объектом *Connection* не происходит. При исполнении команды, вызванной методом *Fill*, соединение открывается только на время извлечения данных, после чего сразу же закрывается. Таким образом, после извлечения данные становятся отсоединенными и ими можно манипулировать в коде независимо от БД, а при необходимости ее можно и обновить.

▶ **Извлечение данных с помощью объекта *DataAdapter***

1. Создайте объект *DataAdapter* соответствующего типа, определяющий подлежащие выборке данные.
2. Создайте экземпляр класса *DataSet* или *DataTable*.
3. Вызовите метод *Fill* объекта *DataAdapter*, передав ему созданный объект *Data Table* или *DataSet*.

Обычно для каждой таблицы с данными создают отдельный объект *DataAdapter*. Если нужно загрузить содержимое нескольких таблиц в один объект *DataSet*, следует задействовать несколько объектов ***DataAdapter***. Один и тот же объект *DataSet* может быть многократно передан методу *Fill*, при вызове которого у каждого объекта *DataAdapter* создается новый объект *DataTable*, заполняется данными и добавляется к объекту *DataSet*.

▶ **Как заполнить объект DataSet данными из нескольких таблиц**

1. Создайте экземпляр класса *DataSet*.
2. Создайте отдельный экземпляр *DataAdapter* соответствующего типа для каждой из таблиц, содержимое которой должно быть в *DataSet*.
3. Вызовите метод *Filly* каждого объекта *DataAdapter*, передав ему созданный ранее объект *DataSet*.

Примечание При добавлении к объекту *DataSet* связанных таблиц *DataAdapter* не воспроизводит связи между ними, поэтому в *DataSet* их придется восстановить вручную путем создания новых объектов *DataRelation*. Об этом рассказано на занятии 3.

Предварительный просмотр данных

Для предварительного просмотра данных, возвращаемых объектами *DataAdapter*, следует выбрать в меню **Data** элемент Preview Data. Откроется окно Data Adapter Preview, показанное на рис. 6-3.

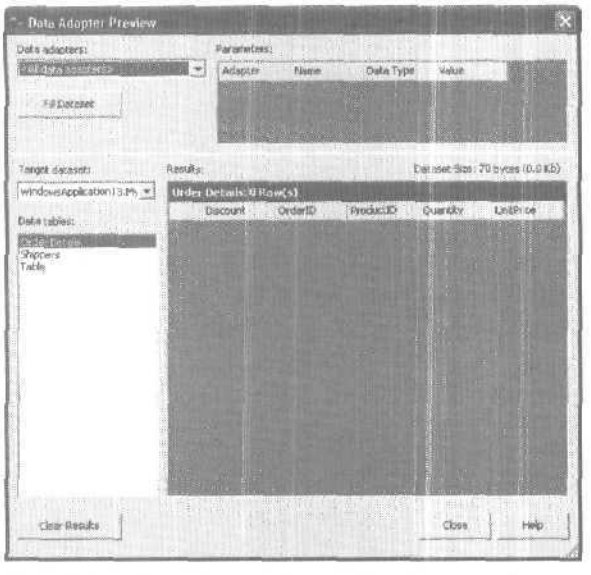


Рис. 6-3. Окно Data Adapter Preview

Здесь вы можете убедиться, что объекты *DataAdapter* правильно сконфигурированы и возвращают корректные данные. Доступные объекты *DataAdapter* отображаются в раскрывающемся списке Data Adapter, из которого можно выбрать один, несколько или все объекты для просмотра. Доступные объекты *DataSet* показаны в раскрывающемся меню Target dataset, откуда объект *DataSet* выбирают для заполнения. Щелкнув кнопку Fill Dataset, можно исполнить метод *Fill* выбранного объекта (или объектов) *DataAdapter,* при этом результаты отображаются в таблице Results, а параметры — в таблице Parameters.

Типизированные объекты DataSet

Стандартные объекты *DataSet* в силу своей конструкции не являются строго типизированными. Каждый элемент данных доступен в форме объекта. Поэтому, чтобы выполнить над ним любые манипуляции, специфичные для некоторого типа, этот объект следует преобразовать в соответствующий тип. При использовании нестрого типизированных переменных вероятны ошибки из-за несоответствия типов, которые весьма непросто найти и устранить. ADO.NET поддерживает типизированный объект *DataSet* ~ альтернативу применению нестрого типизированных переменных.

У типизированного объекта *DataSet,* как можно предположить по его названию, все члены строго типизированы. На таблицы и поля такого объекта *DataSet* можно ссылаться по их дружественным именам, представляющим реальные имена таблиц и столбцов, с которыми вы работаете; их значения доступны в виде значений соответствующих типов, а не объектов. Это дает приложению целый ряд преимуществ. Во-первых, кол программы становится более понятным и его удобнее сопровождать. Во-вторых, ошибки из-за несоответствия типов обнаруживаются в период компиляции, а не в период выполнения — это экономит время, необходимое для тестирования. Наконец, полные имена членов наборов допустимо заменять их дружественными именами, при этом в период разработки имена типизированных членов данных отображаются в окнах среды разработки благодаря технологии Intellisense. Ниже приводится пример эквивалентных строк кода, использующих типизированные и нетипизированные объекты *DataSet.* Оба примера возвращают значения поля Orderl D первой записи таблицы Orders в результирующем наборе *dsorders.* Вот первый пример, использующий нетипизированные объекты *DataSet.*

Visual Basic .NET

```
Dim myOrder As String
myOrder = CType(dsOrders.Tables("Orders").Rows(0).Item("OrderID"), _
   String)
```

Visual C#

```
string myOrder;
myOrder - (string)dsOrders.Tables["Orders"].Rows[0]["OrderID"];
```

Следующий пример написан с применением типизированных объектов *DataSet.* Как видите, второй фрагмент существенно проще и не требует явного преобразования типов.

Visual Basic .NET

```
Dim myOrder As String
myOrder = dsOrders.Orders(0).OrderID
```

Visual C#

```
string myOrder;
myOrder = dsOrders.Orders[0].OrderID;
```

В действительности типизированный *DataSet* — это экземпляр совсем другого класса, производного от *DataSet.* Структура этого класса определяется файлом схемы XML (XSD-файлом), в котором описаны различные особенности структуры объекта *DataSet,* в том числе имена таблиц и столбцов. Для создания нового типизированного объекта *DataSet* требуется файл схемы, но его удается создать, только если структура данных, с которыми вам предстоит работать, известна заранее.

Чтобы генерировать типизированный объект *DataSet,* выберите в диалоговом окне Data команду Generate Dataset — откроется одноименное диалоговое окно, показанное на рис. 6-4.

В открывшемся окне можно выбрать новый объект *DataSet* либо повторно сгенерировать существующий. В этом списке отображаются все доступные таблицы с

их объектами *DataAdapter*. Пометив флажок, расположенный внизу диалогового окна, вы добавите экземпляр нового объекта *DataSet* в окно дизайнера.

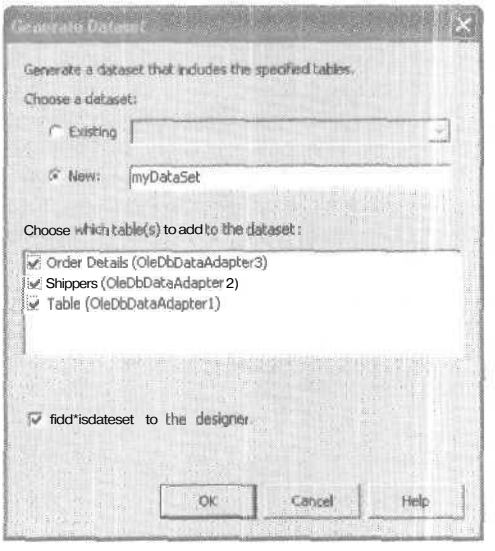


Рис. 6-4. Диалоговое окно Generate Dataset

По щелчку кнопки OK генерируется XSD-файл с корректной схемой, необходимый для создания объекта *DataSet*. Если установить этот флажок, в окно конструктора добавляется экземпляр объекта *DataSet*, созданный на основе генерированной схемы. При заполнении нового объекта *DataSet* имена таблицы и столбцов совпадают с именами, указанными в схеме, а типы данных соответствуют типам, которые возвращаются объектом *DataAdapter*.

▶ **Как сгенерировать строго типизированный объект *DataSet***

1. Создайте и сконфигурируйте объекты *DataAdapter* соответствующего типа, возвращающие нужные таблицы с данными.
2. В меню Data выберите команду Generate Dataset.

Примечание Это также можно сделать, щелкнув правой кнопкой мыши объект *DataAdapter* в окне дизайнера и выбрав команду Generate Dataset либо щелкнув кнопку Generate Dataset в окне Properties.

---

3. В диалоговом окне Generate Dataset щелкните New и назовите новый объект *DataSet*.
4. Выберите в списке доступных таблиц те, которые должны быть включены в схему. Чтобы добавить в окно дизайнера экземпляр объекта *DataSet*, установите соответствующий флажок.
5. Щелкните OK — XSD-файл будет генерирован и добавлен к проекту. Если на предыдущем этапе вы установили флажок, на основе этого файла создается экземпляр объекта *DataSet* и добавляется в окно конструктора.

Чтобы заполнить строго типизированный объект *DataSet*, вызовите метод *Fill* каждого объекта *DataAdapter*, чьи данные вы хотите добавить в объект *DataSet*, передав этот *DataSet* при вызове как целевой объект.

Резюме

- Объект *Connection* позволяет подключиться к БД, Чтобы создать этот объект, достаточно перетащить соответствующий элемент из окна Server Explorer в окно дизайнера или создать новый объект *Connection* в коде.
- **Объект** *Command* представляет команду **SQL** или ссылку на хранимую процедуру. Объекты *OleDbCommand* и *SqlCommand* поддерживают три метода для исполнения команд:
  - *ExecuteNonQuery*
  - *ExecuteScalar*
  - *ExecuteReader*
- Параметры — это значения, необходимые для исполнения команд, которые представлены объектами *Command*. При использовании объекта *OleDbCommand* поля для подстановки значений параметров в операторах SQL обозначаются знаком вопроса. Объект *SqlCommand* допускает применение только именованных параметров.
- Объекты *DataReader* предоставляют доступ к подсоединенным данным. Данные, полученные через объект *DataReader,* доступны только для последовательного однонаправленного чтения. *DataReader* требует монопольного доступа к соединению с источником данных.
- Объекты *DataReader* поддерживают методы для извлечения строго типизированных данных.
- Объекты *DataAdapter* осуществляют взаимодействие между БД и объектом *DataSet,* управляя исполнением команд для заполнения *DataSet* данными из БД и обновления БД содержимым объекта *DataSet.*
- Типизированные объекты *DataSet* — это экземпляры классов, производных от *DataSet.* Эти объекты основаны на схеме XML, они предоставляют строго типизированные данные. На таблицы и поля таких объектов *DataSet* разрешается ссылаться по их дружественным именам.

Занятие 3. Применение объектов DataSet и обновление данных

Объект *DataSet* — это центральный компонент архитектуры доступа, основанной на использовании отсоединенных данных. В архитектуре доступа к данным ADO.NET объекты *DataSet* представляют данные, загруженные в память. Чтобы заполнить *DataSet*, можно воспользоваться объектом *DataAdapter* либо создать новый объект *DataSet,* прочитав данные из файла в формате XML или из «плоского» файла. Объекты *DataSet* также создают программно. Данные этого объекта разрешается связывать с элементами пользовательского интерфейса, а также модифицировать и применять для обновления БД, выполняемого с помощью объекта *DataAdapter.* На этом занятии вы научитесь создавать объекты *DataSet* программно, добавлять к ним объекты *DataRelation* и обновлять БД содержимым объекта *DataSet.*

Изучив материал этого занятия, вы сможете:

- ✓ создать объект *DataSet* программно или на основе данных, прочитанных из плоского файла;
- ✓ добавлять к *DataSet* объекты *Constraints* и *DataRelation;*
- ✓ обновлять БД при помощи объекта *DataAdapter,*
- ✓ применять транзакции БД;
- ✓ разработать схему обработки ошибок, возникающих при обновлении БД.

Продолжительность занятия - около 60 минут.

Объект *DataSet* представляет данные, загруженные в память. В сущности, это полная или частичная копия БД, расположенная на клиентском компьютере. Получение доступа к данным объекта *DataSet* не требует взаимодействия с БД, а их модификация не отражается на содержимом БД вплоть до ее обновления.

На предыдущем занятии вы научились заполнять *DataSet* данными из объекта *DataAdapter.* Это основной способ создания объектов *DataSet,* но иногда требуется доступ к данным, хранящимся иным способом, например в виде массива или текстового файла с разделителями-запятыми. На этом занятии вы узнаете, как это делается.

Создание и заполнение DataSet без использования DataAdapter

Объекты *DataSet* создают программно, как и любые другие объекты, например:

Visual Basic .NET

```
Din myDataSet As New DataSet()
```

Visual C#

```
DataSet myDataSet = new DataSet();
```

В данном примере создается пустой объект *DataSet.* Чтобы заполнить его, необходимо создать новый объект *DataTable* и добавить его к набору *Tables:*

Visual Basic .NET

```
Dim myTable As New DataTable()
myDataSet. Tables. Add(myTable)
```

Visual C#

```
DataTable myTable = new DataTableO;
myDataSet. Tables. Add (myTable);
```

Теперь в наборе *Tables* нового объекта *DataSet* содержится единственная пустая таблица. Можно программно добавить к набору Tables объекта *DataSet* дополнительные объекты *DataTable,*

Чтобы заполнить *DataSet,* необходимо определить структуру его таблиц. Для этого следует программными средствами создать столбцы и добавить их к набору *Columns* объекта *DataTable.* Имя столбца можно указать при создании нового объекта *DataColumn*, например, так:

Visual Basic .NET

```
Dim AccountsColumn As New DataColumn("Accounts")
myDataSet.Tables(0).Columns.Add(AccountsColumn)
```

Visual C#

```
DataColumn AccountsColumn = new DataColumn("Accounts");
myDataSet.Tables[0].Columns.Add(AccountsColumn);
```

Для каждого поля данных, которое предполагается хранить в объекте *DataTable*, требуется отдельный столбец. Создав все необходимые столбцы, можно добавить к *DataTable* объекты *DataRow. DataRow* — это набор, представляющий строку таблицы БД. Создавать экземпляры *DataRow* напрямую нельзя, вместо этого следует вызвать метод *NewRow* объекта *DataTable,* к которому добавляется объект *DataRow.*

Visual Basic .NET

```
Dim myRow As DataRow
myRow = myDataSet.Tables(0).NewRow()
```

Visual C#

```
DataRow myRow;
myRow = myDataSet.Tables[0].NewRow();
```

Созданный объект *DataRow* можно заполнить данными из любого источника, в том числе элементами набора или массива, а также введенными пользователем значениями. Следующий пример демонстрирует заполнение объекта *DataRow* членами набора строк:

Visual Basic .NET

```
' Этот пример предполагает наличие объекта ArrayList с именем StringCollection
Dim Counter As Integer
For Counter = 0 to StringCollection.Count -1
```

```
    myRow.Item(Counter) - StringCollection(Counter)
Next
```

Visual C#

```
// Этот пример предполагает наличие объекта ArrayList с именем StringCollection
for (int i = 0; i < StringCollection.Count; i ++)
\
    myRow.Item[Counter] = StringCollection[i];
}
```

Заполненный объект *DataRow* можно добавить к набору *Rows* объекта *DataTable:*

Visual Basic .NET

```
myDataSet.Tables(0).Rows.Add(myRow)
```

Visual C#

```
myDataSet.Tables[0].Rows.Add(myRow);
```

Следующие действия повторяют, пока не создадут и не заполнят все члены объекта *DataSet.*

▶ Как программно создать объект *DataSet*

1. Объявите и создайте экземпляр класса *DataSet.*
2. Объявите, создайте и добавьте к набору *Tables* объекта *DataSet* нужное число объектов *DataTable.*
3. Для каждого объекта *DataTable* объявите и создайте экземпляры объектов *DataColumn* по числу столбцов таблицы; добавьте эти объекты к набору *Columns* объекта *DataTable.*
4. Создайте новую строку при помощи метода *DataTable. New Row.*
5. Заполните новую строку данными.
6. Добавьте новую строку к набору *Rows* объекта *DataTable.*
7. Повторяйте пункты 4—6, пока не добавите все данные.

Доступ к «плоским» файлам

Многие унаследованные приложения хранят данные в текстовых файлах, поэтому может потребоваться возможность чтения такого файла в объект *DataSet.* Общий подход, описанный в предыдущем разделе, позволяет читать данные из текстовых файлов и записывать их в объект *DataSet* во время выполнения. Члены пространства имен *System. IO* упрощают доступ к текстовым файлам, а методы класса *System.String* позволяют выполнять их синтаксический разбор.

При чтении данных из текстового файла прежде всего необходимо определить формат данных. Обычно значения, хранящиеся в текстовых файлах, разделены стандартным символом — разделителем. Это может быть запятая, двоеточие, точка с запятой или что-то еще. Данные часто хранятся в виде текстовых строк, заканчивающихся символом возврата каретки, обозначающим начало новой строки. Число полей в каждой строке должно совпадать с числом столбцов таблицы.

Определив формат, в котором хранятся данные, пора приступать к конструированию объекта *DataSet.* Способы программного создания объекта *DataSet* описаны в предыдущем разделе. Необходимо создать по одному объекту *DataTable* на каж-

дую таблицу с данными. Далее к объекту *DataTable* следует добавить соответствующее число столбцов. Иногда имена столбцов берут из первой строки текстового файла. Для этого следует прочитать значения из первой строки текстового файла и программными средствами назначить столбцам соответствующие имена.

Для чтения текстового файл используют класс *System.IO.StreamReader*, который способен открыть файл и вернуть содержащиеся в нем символы. Открыв файл, можно воспользоваться классом *System.String*, чтобы выделить элементы данных и записать их в новые строки.

Естественно, формат хранения данных в «плоских» файлах может отличаться, поэтому подходящий способ доступа к данным необходимо выбрать в каждом конкретном случае. Следующий пример демонстрирует типичную ситуацию, в которой требуется получить доступ к данным текстового файла, где записи представлены строками, а поля разделены запятыми.

Visual Basic .NET

```
' Этот пример предполагает наличие текстового файла myFile.txt.
' содержащего неизвестное число строк, каждая из которых состоит
' из семи полей.
' Создать новый объект DataSet.
Dim myDataSet As New DataSet()
' Создать новый объект DataTable и добавить его к набору Tables,
Dim aTable As New DataTable("Table 1")
myDataSet.Tables.Add("Table 1")
' Создать семь столбцов, присвоить им имена и добавить их к Table 1,
Dim Counter As Integer
Dim aColumn As DataColumn
For Counter = 0 to 6
   aColurnn = New DataColumn("Column " & Counter.ToString())
   myDataSet.Tables("Table 1").Columns.Add(aColumn)
Next
' Создать объект StreamReader для чтения файла и строковую переменную для
' хранения выходных данных объекта StreamReader.
Dim myReader As New System.IO.StreamReader("C:\myFile.txt")
Dim mystring As String
' Проверить, достиг ли StreamReader конца потока.
While myReader.Peek <> -1
   ' Прочитать строку данных из текстового файла.
   mystring = myReader.ReadLine
   ' Вызвать метод String.Split, чтобы создать массив строк, представляющий
   ' все элементы строки, и добавить этот массив к Table 1 как новый объект
   ' DataRow.
   myDataSet.Tables("Table 1").Rows.Add(mystring.Split(", "c))
End While
```

Visual C#

```
// Этот пример предполагает наличие текстового файла myFile.txt,
// содержащего неизвестное число строк, каждая из которых состоит
```

```
If из семи полей.
// Создать новый объект DataSet.
DataSet myDataSet = new DataSet();
// Создать новый объект DataTable и добавить его к набору Tables.
myDataSet.Tables.Add("Table 1");
/7 Создать семь столбцов, присвоить им имена и добавить их к Table 1.
DataColumn aColumn;
for (int counter - 0; counter < 7; counter ++)
{
   aColumn = new DataColumn("Column " + counter.ToString());
   myDataSet.Tables["Table 1"].Columns.Add(aColumn);
}
// Создать объект StreamReader для чтения файла и строковую переменную для
// хранения выходных данных объекта StreamReader.
System.IO.StreamReader myReader - new
   System.IO.StreamReader("C:\\myFile.txt");
stringmyString;
// Проверить, достиг ли StreamReader конца потока.
while (myReader.Peek() != -1)
{
   // Считывает строку данных из текстового файла
   myString = myReader.ReadLine();
   If Вызвать метод String.Split, чтобы создать массив строк,
   // представляющий все элементы строки, и добавить этот массив к Table 1
   /7 как новый объект DataRow.
   myDataSet.Tables["Table 1"].
      Rows.Add(myString.Split(char.Parse(",")));
}
```

Объект DataRelation

Объект *DataRelation* представляет связь между двумя полями разных таблиц. Предположим, в каждой из двух таблиц — Customers и Orders — содержится поле CustomerID. В таблице Customers любой покупатель встречается только один раз, а в таблице Orders он может встречаться несколько раз. Сопоставив значения полей CustomerID в таблицах Orders и Customers, удается определить покупателя, разместившего заказ. Поле CustomerID в таблице Orders напрямую ссылается на строку со сведениями о покупателе в таблице Customers и может служить ключом для извлечения этой строки; аналогичное поле таблицы Customers предназначено для извлечения любых строк из таблицы Orders. Это типичный пример связи типа *«один ко многим»*, когда идентификатор, уникальный в одной таблице, соответствует нескольким строкам в другой таблице. Такой тип связи встречается чаще всего.

Объект *DataRelation* позволяет создать подобную связь между двумя таблицами объекта *DataSet*. Объекты *DataRelation,* представляющие связи между таблицами объекта *DataSet*, хранятся в его свойстве *Relations*. При создании объекта *DataRelation* необходимо указать его имя, а также имена родительского и дочернего полей. В случае типизированного объекта *DataSet* у этих полей должен быть одинаковый тип. Вот пример создания объекта *DataRelation*:

Visual Basic .NET

```
Dim myRelation As New _
   DataRelation("Data Relation 1", column1, column2)
```

Visual C#

```
DataRelation myRelation -
   new DataRelation("Data Relation 1", columnI,
   column2);
```

Чтобы активизировать созданный объект *DataRelation*, следует добавить его к набору *Relations* объекта *DataSet*:

Visual Basic .NET

```
myDataSet.Relations.Add(myRelation)
```

Visual C#

```
myDataSet.Relations.Add(myRelation);
```

▶ **Создание нового объекта *DataRelation***

1. Объявите и создайте новый экземпляр класса *DataRelation*, ссылающийся на родительское и дочернее поля. Эти поля должны находиться в таблицах, расположенных в одном и том же объекте *DataSet*. При использовании типизированного объекта *DataSet* поля также должны быть одинакового типа.
2. Добавьте созданный объект *DataRelation* к набору *Relations* объекта *DataSet*.

Извлечение связанных записей

Объект *DataRelation* позволяет извлечь строки, содержащие родительские и дочерние поля. Для этого нужно вызвать методы *GetChildRows* или *GetParentRow* объекта *DataRow*, которые требуют передать им действительный объект *DataRelation*. Метод *GetChildRow* возвращает массив дочерних строк, заданных объектом *DataRelation*. Метод *GetParentRow* отличается от него лишь тем, что возвращает только одну строку — родительскую, также заданную объектом *DataRelation*. Вызов этих методов иллюстрируется следующим примером, который предполагает наличие таблиц Customers и Orders, связанных посредством объекта *DataRelation* с именем *CustomersOrders*; обе таблицы располагаются в объекте *DataSet* с именем *myDataSet*.

Visual Basic .NET

```
Dim ChildRows() As DataRow
Dim ParentRow As DataRow
' Вернуть все дочерние строки первой строки таблицы Customers,
' заданные объектом DataRelation с именем CustomersOrders.
ChildRows = _
   myDataSet.Tables("Customers").Rows(1).GetChildRows(CustomersOrders)
' Вернуть дочернюю строку пятой строки таблицы Orders,
' заданную объектом DataRelation с именем CustomersOrders.
ParentRow = _
   myDataSet.Tables("Orders").Rows(5).GetParentRow(CustomersOrders)
```

Visual C#

```
DataRow[] ChildRows;
DataRow ParentRow;
// Вернуть все дочерние строки первой строки таблицы Customers,
// заданные объектом DataRelation с именем CustomersOrders.
ChildRows =
   myDataSet.Tables["Customers"].Rows[1].GetChildRows(CustomersOrders);
// Вернуть дочернюю строку пятой строки таблицы Orders,
// заданную объектом DataRelation с именем CustomersOrders.
ParentRow =
   myDataSet.Tables["Orders"].Rows[5].GetParentRow(CustomersOrders);
```

Ограничения

Объекты *DataRelation* тесно связаны с ограничениями — они определяют правила вставки строк в *DataTable* и управления содержимым этих объектов. Ограничения бывают двух типов — *UniqueConstraint* и *ForeignKeyConstraint*.

Ограничение *UniqueConstraint* запрещает добавлять в таблицу дублирующиеся элементы, обычно с этим ограничением определяют первичный ключ таблицы. *ForeignKeyConstraint* определяет правила обновления дочерних строк при изменении родительской строки.

В общем случае ограничение создают так: сначала генерируют экземпляр объекта, представляющего ограничение соответствующего типа, и добавляют созданный объект к набору *Constraints* таблицы, на поле которого налагается ограничение.

Ограничение действует, только если значение свойства *EnforceConstraints* объекта *DataSet* установлено в *true*.

Ограничения Unique

Создать ограничение *UniqueConstraint* несложно. Проще всего это сделать, установив свойство *Unique* объекта *DataColumn* в *True (true),* например:

Visual Basic .NET

```
myDataColumn.Unique = True
```

Visual C#

```
myDataColumn.Unique = true;
```

В результате неявно создается ограничение *UniqueConstraint,* которое и добавляется к набору *Constraints.* Это ограничение можно создать и явно: достаточно указать поле (столбец), в котором должны быть только уникальные значения. Созданное таким образом ограничение затем придется вручную добавлять к набору *Constraints:*

Visual Basic .NET

```
Dim myConstraint As New UniqueConstraint(myDataRow)
myDataTable.Constraints.Add(myConstraint)
```

Visual C#

```
UniqueConstraint myConstraint - new UniqueConstraint(myDataRow);
myDataTable.Constraints.Add(myConstraint);
```

Ограничения Unique позволяют также определить составной ключ. Предположим, имеется таблица Employees со столбцами FirstName и LastName; в первом из них могут быть дублирующиеся элементы, а во втором — нет, но комбинации значений этих столбцов должны быть уникальными. В этой ситуации ограничение *UniqueConstraint* позволяет задать массив столбцов, комбинации значений которых будут уникальными, например:

Visual Basic .NET

```
Dim myColumns(1) As DataColumn
myColumns(0) - EmployeesTable.Columns("FirstName")
myColumns(1) = EmployeesTable.Columns("LastName")
Dim myConstraint As New UniqueConstraint(myColumns)
EmployeesTable.Constraints.Add(myConstraint)
```

Visual C#

```
DataColumn[] myColumns = new DataColumn[2];
myColumns[0] = EmployeesTable.Columns["FirstName"];
myColumns[1] = EmployeesTable.Columns["LastName"];
UniqueConstraint myConstraint = new UniqueConstraint(myColumns);
EmployeesTable.Constraints.Add(myConstraint);
```

Ограничения Foreign Key

Ограничение *ForeignKeyConstraint* (внешний ключ) определяет изменение дочерних строк при модификации или удалении родительской строки. Чтобы создать объект *ForeignKeyConstraint*, необходимо указать родительское и дочернее поле, например:

Visual Basic .NET

```
Dim myConstraint As New _
   ForeignKeyConstraint(CustomersTbl.Columns("CustomerID"), _
   OrdersTbl.Columns("CustomerID"))
```

Visual C#

```
ForeignKeyConstraint myConstraint = new
   ForeignKeyConstraint(CustomersTbl.Columns["CustomerID"],
   OrdersTbl.Columns["CustomerID"]);
```

Подобно *UniqueConstraint,* для активации ограничение *ForeignKeyConstraint* необходимо добавить к набору *Constraints* таблицы, содержащей родительское поле:

Visual Basic .NET

```
CustomersTbl.Constraints.Add(myConstraint)
```

Visual C#

```
CustomersTbl.Constraints.Add(myConstraint);
```

Ограничение *ForeignKeyConstraint* определяет три правила, регламентирующих отношения между родительским и дочерним полями:

- *UpdateRule* — применяется при модификации родительской строки;
- *DeleteRule* — применяется при удалении родительской строки;
- *AcceptRejectRule* — применяется каждый раз при вызове метода *AcceptChanges* объекта *DataTable*, для которого определено ограничение.

Эти правила представлены свойствами объекта ограничения, которые могут принимать значения, описанные в таблице 6-2.

Таблица 6-2. Свойства объекта ограничения и их возможные значения

Значение	Описание
Cascade	Модификации родительской строки реплицируются в дочерние строки
None	Модификации родительской строки не отражаются на дочерних строках; в результате могут появиться дочерние строки со ссылками на недействительные родительские строки
SetDefault	Полю внешнего ключа в связанных дочерних записях присваивается значение по умолчанию (заданное свойством *DefaultValue* этого ключа)
SetNull	Полю внешнего ключа в дочерней таблице присваивается значение *DBNull*; в результате в дочерней таблице могут появиться недействительные значения

Примечание Свойство *AcceptRejectRule* принимает значение *Cascade* либо *None*,

Всем свойствам, представляющим правила, по умолчанию присвоено значение *Cascade*.

Модификация и обновление данных

Содержимым объекта *DataSet* можно манипулировать, в частности редактировать его на клиентской машине. Разрешается модифицировать значения существующих объектов *DataRow*, а также удалять и добавлять к *DataSet* новые объекты *DataRow*. Однако модификации не будут реплицированы в БД, пока вы не обновите се с помощью объекта *DataAdapter*.

Каждый экземпляр *DataSet* поддерживает две версии этого объекта: текущую, расположенную на клиентском компьютере и содержащую все модификации, и исходную, не менявшуюся со времени заполнения *DataSet*. При вызове метода *Update* объекта *DataAdapter* на основе исходных значений генерируются команды UPDATE, INSERT и DELETE, необходимые для обновления БД.

Модификация данных

Есть два способа модификации объекта *DataSet:* через пользовательский интерфейс – – при помощи элементов управления, связанных с данными, и программными средствами. Изменения, внесенные через связанные с данными элементы управления, автоматически вносятся в соответствующую строку.

Примечание Связывание данных подробнее обсуждается на следующем занятии.

Также поддерживается программное добавление данных к объекту *DataRow* путем установки свойства *Item* объекта *DataRow,* представляющего значения полей строки. Вот как это делается:

Visual Basic .NET

```
' Item - это свойство по умолчанию объекта DataRow, поэтому
' ссылаться на него явно не обязательно, Получить доступ к полю
' строки можно, ссылаясь на него по индексу, либо по имени,
myDataRow(2) = "Splunge"
myDataRow("Customers") = "Winthrop"
```

Visual C#

```
// Item - это свойство по умолчанию объекта DataRow. поэтому
// ссылаться на него явно не обязательно. Получить доступ к полю
// строки можно, ссылаясь на него по индексу, либо по имени.
myDataRow[2] = "Splunge";
myDataRow["Customers"] = "Winthrop";
```

Каждый объект *DataRow* поддерживает две версии состояния: исходную, содержащую первоначальные значения *DataRow,* и модифицированную. В любой момент можно отменить (откатить) любые изменения, внесенные в объект *DataRow,* вызвав метод *RejectChanges*:

Visual Basic .NET

```
myDataRow.RejectChanges()
```

Visual C#

```
myDataRow.RejectChanges();
```

Чтобы внести некоторые изменения в объект *DataRow,* вызовите метод *AcceptChanges.* В результате модифицированная версия объекта *DataRow* записывается поверх исходной.

Visual Basic .NET

```
myDataRow.AcceptChanges()
```

Visual C#

```
myDataRow.AcceptChanges();
```

Внимание! Если планируется реплицировать модификации в источник данных с помощью объекта *DataAdapter,* вызывать метод *AcceptChanges* до вызова метода *DataAdapter. Update* нельзя, иначе объекту *DataAdapter* не удастся получить доступ к исходной версии данных и сгенерировать корректную команду **UPDATE.**

Определить состояние объекта *DataRow* можно посредством свойства *RowState.* Оно принимает одно из значений, описанных в таблице 6-3.

Объекты *DataTable* и *DataSet* также поддерживают методы *RejectChanges* и *AcceptChanges*, позволяющие соответственно отвергнуть или принять все изменения, внесенные в *DataTable* или *DataSet*.

Таблица 6-3. Значения свойства *RowState*

Значение	Описание
Unchanged	Строка осталась в исходном виде либо не была модифицирована после вызова *AcceptChanges*
Modified	Строка модифицирована после последнего вызова *AcceptChanges*
Added	Строка создана и добавлена в *DataRowCollection*, но метод *AcceptChanges* еще не вызван
Deleted	Строка удалена с помощью метода *DataRow.Delete*
Detached	Строка создана, но не добавлена ни в один из наборов *DataRowCollection*

Обновление базы данных

Закончив манипуляции над клиентской копией данных, можно обновить БД, вызвав метод *Update* у каждого объекта *DataAdapter*.

Visual Basic .NET

```
myDataAdapter.Update()
myOtherDataAdapter.Update()
```

Visual C#

```
myDataAdapter.Update();
myOtherDataAdapter.Update();
```

В результате изменения, внесенные в клиентскую копию данных, копируются в БД. При этом можно указать объект *DataSet*, *DataTable* или массив объектов *DataRow*s, подлежащий обновлению:

Visual Basic .NET

```
myDataAdapter.Update(myDataSet)
myDataAdapter.Update(myDataTable)
myDataAdapter.Update(myDataRows)
```

Visual C#

```
myDataAdapter.Update(myDataSet);
myDataAdapter.Update(myDataTable);
myDataAdapter.Update(myDataRows);
```

Транзакции

В некоторых случаях требуется выполнить серию обновлений так, чтобы все они завершились успехом либо неудачей. Хороший пример приложения, где это необходимо, — программа для банковского учета. Давайте рассмотрим метод, который

снимает определенную сумму с одного счета (первая таблица) и зачисляет ее на другой счет (вторая таблица). Если первая операция пройдет успешно, а вторая окончится неудачей, результат окажется печальным. Решать подобные проблемы позволяют транзакции. *Транзакция* (transaction) — это набор логически связанных действий, выполняемых как единая операция. Для успеха транзакции все составляющие ее действия должны завершиться успешно, в противном случае выполняется откат всех операций транзакции, выполненных к этому моменту.

Инициировать транзакцию можно, вызвав метод *Begin Transaction* любого объекта *Connection,* представляющего открытое соединение. Этот метод возвращает ссылку на транзакцию, которую нужно записать в свойство *Transaction* объектов всех команд, участвующих в данной транзакции. Далее необходимо исполнить команды транзакции. Если все они завершатся удовлетворительно, вызывается метод *Transaction. Commit,* что подтверждает изменения БД, в противном случае вызывается метод *Transaction.Rollback*, который отменяет коррективы. Следующий пример демонстрирует исполнение транзакции с помощью объекта *OleDbConnection* с именем *myConnection* и двух объектов *OleDbCommand* с именами *Update1* и *Update2.*

Примечание Для использования транзакций источник данных должен поддерживать транзакции.

Visual Basic .NET

```
' Транзакции необходимо заключить в блок Try..Catch..Finally,
' чтобы иметь возможность перехвата любых исключений.
Dim myTransaction As System.Data.OleDb.OleDbTransaction
Try
   myConnection.Open()
   ' Создать новый объект транзакции и записать его в переменную myTransaction.
   myTransaction = myConnection.BeginTransaction
   ' Добавить команды Update1 и Update2 к транзакции.
   Update1.Transaction = myTransaction
   Update2.Transaction = myTransaction
   ' Исполнить команды Update1 и Update2.
   Update1.ExecuteNonQuery()
   Update2.ExecuteNonQuery()
   ' Если исключений не возникло, считать транзакцию успешной,
   myTransaction.Commit()
Catch ex As Exception
   ' в противном случае отменить транзакцию.
   myTransaction.Rollback()
Finally
   ' В любом случае соединение необходимо закрыть.
   myConnection.Close()
End Try
```

Visual C#

```
// Транзакции необходимо заключить в блок Try..Catch..Finally,
// чтобы иметь возможность перехвата любых исключений.
```

```
System.Data.OleDb.OleDbTransaction myTransaction = null;
try
{
   myConnection.Open();
   // Создать новый объект транзакции и записать его а переменную myTransaction.
   myTransaction = myConnection.BeginTransaction();
   // Добавить команды Update1 и Update2 к транзакции.
   Update1.Transaction= rnyTransaction;
   Update2.Transaction - myTransaction;
   // Исполнить команды Update1 и Update2.
   Update1.ExecuteNonQuery();
   Update2. ExecuteNonQuery0;
   // Если исключений не возникло, считать транзакцию успешной,
   myTransaction.Commit();
!
catch (Exception ex)
{
   // в противном случае отменить транзакцию.
   myTransaction.Rollback();
}
finally
!
   // В любом случае соединение необходимо закрыть.
   myConnection.Close();
}
```

► **Исполнение нескольких команд как транзакции**

1. Откройте соединение с БД.
2. Получите ссылку на транзакцию, вызвав метол *Connection.BeginTransaction.*
3. Поместите полученную ссылку в свойство *Transaction* всех команд-участников транзакции.
4. Выполните команды транзакции.
5. Вызовите метод *Transaction. Commit,* чтобы завершить транзакцию.

Транзакции доступны и при взаимодействии с БД через объект *DataAdapter,* но в этом случае применять их сложнее. Прежде всего необходимо создать транзакцию, вызвав метод *BeginTransaction* объекта, который представляет открытое соединение. Далее созданную транзакцию помещают в свойства *Transaction* объектов, представляющих команды *InsertCommand, UpdateCommand* и *DeleteCommand* всех объектов *DataAdapter,* которые будут задействованы при обновлении БД. После этого можно вызвать метод *Update* каждого объекта *DataAdapter* и завершить транзакцию, вызвав метод *Commit* или *Rollback* объекта этой транзакции.

Обработка ошибок при обновлении базы данных

Причины ошибок, возникающих при обновлении данных, бывают самыми разными: вставка записи с дублирующимся значением первичного ключа, попытка модификации строк, используемых другим клиентом и т.д. Разрабатывая код для обнов-

ления строк, необходимо предусмотреть любые ошибки и подготовиться к устранению их последствий.

Объекты *SqlDataAdapters* и *OledbDataAdapters* поддерживают событие *RowUpdated*, которое генерируется между попыткой обновления строки и любыми возникающими при этом исключениями. В его обработчик можно поместить код, способный обработать любые ошибки, возникающие при обновлении, не вовлекая в этот процесс блоки обработки исключений, которые используют много ресурсов.

В зависимости от типа применяемого объекта *DataAdapter*, событие *RowUpdated* передает экземпляр класса *SqlRowUpdateEventArgs* или *leDbRowUpdatedEventArgs* Аргументы события *RowUpdated* позволяют определить, какая ошибка возникла и как ее обрабатывать. В таблице 6-4 описаны некоторые свойства классов *SqlRowUpdated-EventArgs* и *OleDbRowUpdatedEventArgs*.

Таблица 6-4. Свойства класса *RowUpdatedEventArgs*

Имя	Описание
Command	Представляет команду, исполняемую при обновлении строки
Errors	Возвращает любые ошибки, генерируемые провайдером данных при исполнении этой команды
RecordsAffected	Возвращает число записей, измененных при исполнении команды, которая задана свойством *Command*
Row	Возвращает обновленную строку
Status	Возвращает значение свойства *UpdateStatus* объекта команды

Чтобы во время обработки события *RowUpdated* определить, произошла ли ошибка, проверьте свойство *Status* объекта аргументов события. Это свойство может принимать следующие четыре значения:

- *Continue* — объект *DataAdapter* продолжит обработку строк. Если ошибки не возникают, свойство *Status* сохраняет это значение;
- *ErrorsOccurred* — при попытке обновления данной строки возникла одна или несколько ошибок;
- *SkipAllRemainingRows* — при обновлении текущую строку и все остальные следует пропустить;
- *SkipCurrentRow* — при обновлении следует пропустить только текущую строку.

При возникновении ошибки свойство *Status* получает значение *ErrorsOccurred,* Обработчик событий *RowUpdated* позволяет обработать возникшую ошибку, изменив значение свойства *Status.* Если свойство *Status* установлено в *SkipCurrentRow,* при обновлении объект *DataAdapter* пропустит строку, вызвавшую ошибку, и перейдет к другим строкам. Если же свойство *Status* установлено в *SkipAllRemaining-Rows, DataAdapter* прекратит обновление. Если *Status=Continue,* объект *DataAdapter* игнорирует ошибку и продолжает обновление. Имейте в виду, что в зависимости от типа источника данных, это может привести к получению непредсказуемых результатов. Если свойство *Status* сохраняет значение *ErrorsOccurred,* генерируется исключение, которое передается обработчику исключений приложения.

Сведения о возникшей ошибке получают с помощью свойства *Errors,* которое возвращает объект исключения, представляющий возникшую ошибку.

Выбор способа обработки ошибок, возникающих при обновлении БД, определяется главным образом особенностями приложения. В одних случаях исключения

лучше передавать для обработки коду приложения, а в других — использовать для этого обработчик события *RowUpdated*. Следующий пример иллюстрирует простую схему обработки ошибок обновления с помощью события *RowUpdated* объекта *SqlDataAdapter*:

Visual Basic .NET

```
Private Sub myDataAdapter_RowUpdated(ByVal sender As Object, _
   ByVal e As System.Data.SqlClient.SqlRowUpdatedEventArgs) _
   Handles myDataAdapter.RowUpdated
   ' Проверить свойство Status объектов аргументов события на наличие ошибок.
   If e.Status = UpdateStatus.ErrorsOccurred then
      ' Уведомить пользователя об ошибке и
      ' предоставить ее описание.
      MessageBox.Show("An error of type " & e.Errors.ToString() & _
         "occurred. Here is some additional information: " & _
         e.Errors.Message)
      ' Пропустить данную строку при обновлении,
      e.Status = UpdateStatus.SkipCurrentRow
   End If
End Sub
```

Visual C#

```
private void myDataAdapter_RowUpdated(object sender,
   System.Data.SqlClient.SqlRowUpdatedEventArgs e)
{
   // Проверить свойство Status объектов аргументов события на наличие ошибок.
   if (e.Status == UpdateStatus.ErrorsOccurred)
   {
      // Уведомить пользователя об ошибке и
      // предоставить ее описание.
      MessageBox.Show("An error of type " + e.Errors.ToString() +
         "occurred. Here is some additional information: " +
         e.Errors.Message);
      // Пропустить данную строку при обновлении.
      e.Status = UpdateStatus.SkipCurrentRow;
   }
}
```

Резюме

- Объекты *DataSet* и *DataTable* разрешается создавать независимо от объектов *DataAdapter* и программно заполнять их.
- Класс *System.IO.StreamReader* позволяет получать доступ к текстовым файлам и считывать их содержимое, а методы класса *String* — выполнять его синтаксический разбор.
- Объекты *DataRelation* представляют связи между столбцами различных таблиц, а также позволяют реализовать ограничения и извлекать связанные строки.
- Ограничения — это правила, регламентирующие заполнение полей. Предусмотрено два вида ограничений: *UniqueConstraint* и *ForeignKeyConstraint*.
- Объект *DataSet* поддерживает две версии хранящихся в нем данных: исходную и модифицированную. При вызове метода *Update* объекта *DataAdapter* генерируются команды для обновления БД на основе сравнения этих версий.
- Чтобы исполнить несколько команд как транзакцию, получите объект *Transaction* для открытого соединения и поместите ссылку на него в свойство *Transaction* объектов *Command*, представляющих команды-участники транзакции. Они будут исполнены по принципу «все или ничего»: либо все они завершатся успехом, либо выполняется откат транзакции.
- Событие *RowUpdated* позволяет обработать ошибки, возникающие при обновлении данных. Чтобы выяснить, возникла ли ошибка, следует проверить свойство *Status* аргумента этого события. Обработать возникшие ошибки можно, установив для этого свойства соответствующее значение.

Занятие 4. Связывание, просмотр и фильтрация данных

Возможность просмотра данных жизненно важна для многих приложений. Связывание данных позволяет ассоциировать записи источника данных с элементами управления формы для просмотра и модификации этих данных. На этом занятии вы научитесь применять ADO.NET для связывания элементов управления с данными, управлять передачей данных в приложении, а также создавать с помощью класса *DataView* представления для объектов *DataTables*, поддерживающие фильтрацию и сортировку, и узнаете, как управлять несколькими объектами *DataView* с помощью одного объекта *DataViewManager*.

Изучив материал этого занятия, вы сможете:

- ✓ описать роль провайдера данных;
- ✓ описать роль потребителя данных;
- ✓ связать свойство объекта-потребителя данных с провайдером данных;
- ✓ управлять потоком данных в приложении Windows Forms;
- ✓ осуществлять фильтрацию и сортировку данных с помощью компонента *DataView*;
- ✓ описать роль класса *DataViewManager* и рассказать, как его применяют.

Продолжительность занятия - около 45 минут.

Связывание данных

Связыванием данных называется создание связи между двумя объектами — провайдером и потребителем данных. Провайдер — это источник данных, связанный с объектом-потребителем данных, который получает данные, после чего обрабатывает либо отображает их. Типичный пример связывания данных — элемент управления *TextBox*, связанный с источником данных. Он отображает значение некоторого поля в текущей строке источника данных, к которому привязан. При модификации этого поля отображаемое элементом управления значение также изменится.

Провайдеры данных

Провайдер данных участвует в связывании, предоставляя значения для связанных с ним свойств и элементов управления.

Примечание В данном контексте термин «провайдер данных» обозначает не многокомпонентный провайдер ADO.NET, применяемый для доступа к отсоединенным данным, а вообще любой объект, с которым может быть связан другой объект — потребитель данных.

В .NET Framework провайдерами данных могут быть любые объекты, в которых реализован интерфейс *IList*, то есть не только объекты ADO.NET, такие, как *DataSet*, *DataTable* и *DataColumn*, но и другие, например массивы и наборы. Объект *DataView*, о котором пойдет речь далее, является настраиваемым представлением данных и способен быть их провайдером.

Другая функция провайдеров — управление потоком данных. В прежних технологиях доступа к данным для управления потоком данных применялись курсоры. При перемещении курсора на другую запись выполнялось обновление связанных элементов управления. Поскольку ADO.NET основана на доступе к отсоединенным данным, концепция курсора в ней отсутствует как таковая. Теперь с каждым источником данных связан объект *CurrencyManager,* который следит за положением текущей записи. Управление объектами *CurrencyManager* осуществляется с помощью объекта *BindingContext* формы. Подробно управление потоком данных описано далее.

Потребители данных

Потребитель данных — это элемент управления, свойства которого привязаны к источнику данных. В .NET Framework с источником данных можно связать любое доступное в период выполнения свойство любого элемента управления. Например, свойство *Size* — с записью БД, каталогом или другим свойством элемента управления, доступным в период выполнения.

Связывание данных применяется очень широко. Типичный пример использования связывания данных — формы для ввода данных, содержащие элементы управления (*TextBox*, *CheckBox*, *ListBox* и т.д.), связанные с соответствующими полями объекта *DataSet*. Новые записи можно ввести вручную через соответствующие элементы управления формы. Готовая запись добавляется к объекту *DataSet,* который используется для обновления БД.

Выделяют два вида связывания данных — простое и сложное. В простом задействовано только одно поле источника данных, последовательно предоставляющее значения связанному с ним компоненту. Например, элемент управления *Label можно* связать с полем объекта *DataTable,* чтобы он отображал текущее значение поля, с которым связан. Сложное связывание позволяет связать несколько записей с одним элементом управления. Так, элементы управления *ListBox* и *ComboBox* можно связать с полем объекта *DataTable* или *DataView* — они будут отображать во время выполнения все значения этого поля, а не только текущее. Обычно сложное связывание применяют для создания элементов управления, отображающих набор значений и позволяющих выбрать определенные строки данных. Такие элементы управления, как *DataGrid,* поддерживают еще более сложный вид связывания: их можно связать со всеми значениями объекта *DataTable, DataView* или даже *DataSet.*

Создание элемента управления, использующего простое связывание данных

Свойство *DataBindings* позволяет создать элементы управления, применяющие простое связывание данных. Это свойство является экземпляром класса *ControlBindingsCollection,* который следит за связыванием элементов управления с источником данных и управляет им. В период разработки свойство *DataBindings* отображается и окне Properties в виде отдельного узла. Раскрыв его, вы получите доступ к списку свойств, которые чаще всего связывают с данными (рис. 6-5).

Чтобы связать такое свойство с источником данных, щелкните поле рядом с именем свойства — откроется список доступных провайдеров данных, где объекты *DataSet* отображаются в виде узлов. Раскрыв такой узел, вы получите список доступных объектов *DataTable,* там же перечислены объекты *DataView.* А раскрыв узел объекта *DataView* — список полей источника данных. Найдите в нем нужный столбец и выберите его — вы создадите связь между выбранным полем и данным свойством элемента управления.

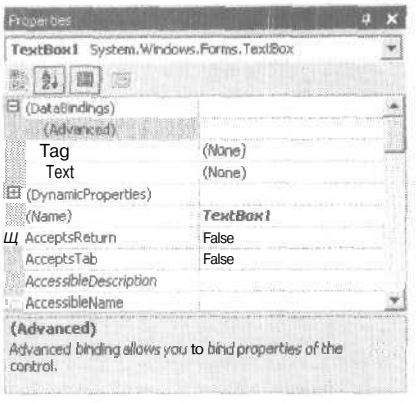


Рис. 6-5. Свойства, которые связывают с данными чаще всего

Так можно связать с источником данных любое свойство элемента управления. Чтобы связать с данными свойство, отсутствующее в списке DataBindings, щелкните поле (Advanced) под узлом DataBindings, затем щелкните кнопку (...), чтобы открыть диалоговое окно Advanced Data Binding, показанное на рис. 6-6.

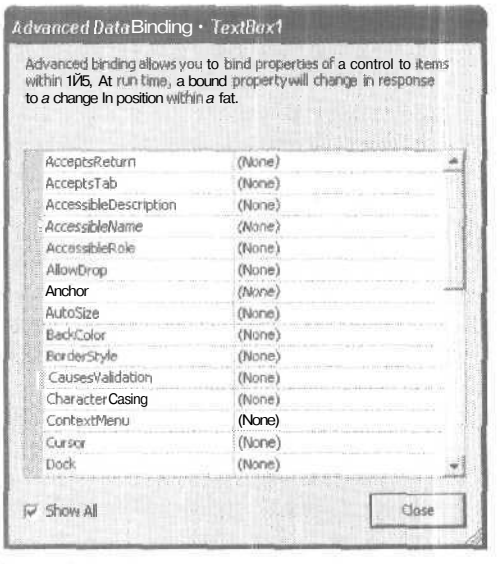


Рис. 6-6. Диалоговое окно Advanced Data Binding

В нем отображается полный список свойств элемента управления, доступных в период выполнения: выбрав здесь нужное поле, вы свяжете его с определенным свойством элемента управления.

▶ **Связывание данных в период разработки**

1. В окне Properties раскройте узел DataBindings.
2. **Если нужное свойство присутствует в списке узла,** выберите в раскрывающемся списке соответствующее поле.

   Если нужное свойство отсутствует в списке узла, щелкните кнопку (...) рядом с полем (Advanced), чтобы вызвать диалоговое окно Advanced Data Binding; ука-

жите нужное свойство, затем выберите в списке рядом с его именем поле, которое вы хотите связать с этим свойством.

Связывание данных в период выполнения

Часто в период выполнения требуется модифицировать источник данных, с которым связан некоторый элемент управления. Также возможна ситуация, когда источник данных, с которым будет связан элемент управления, не известен в период разработки. Наконец, иногда необходимо связать элемент управления с массивом или набором, экземпляр которого разрешается создавать только в период выполнения. В этих случаях следует выполнять связывание данных программными средствами- Для этого воспользуйтесь свойством *DataBindings* элемента управления, которое является экземпляром класса *Control Bindings Collection,* Установив соответствующий член набора *DataBindings,* вы сможете добавить, удалить или сбросить сведения о связанном источнике данных в период выполнения.

Для связывания свойств с источником данных служит метод *DataBindings.Add.* При вызове этого метода создается связь с источником данных и добавляется к набору *DataBindings,* Метод *Add* принимает три параметра: имя свойства (типа String), которое нужно связать с данными, источник данных (объект) и имя члена объекта источника данных, с которым связывают это свойство. Следующий пример демонстрирует связывание свойства *Text* элемента управления *TextBox с* полем CustomerI D таблицы Customers, хранимой в объекте *DataSet* с именем *DataSet1*:

Visual Basic .NET

```
TextBox1.DataBindings.Add("Text", DataSet1.Customers, "CustomerID")
```

Visual C#

```
TextBox1.DataBindings.Add("Text", DataSet1.Customers, "CustomerID");
```

В некоторых случаях необходимо связать объекты, имеющие только один член данных, такой, как наборы и массивы — при этом в третьем параметре необходимо передать пустую строку. Именно так, например, связывают свойство элемента управления с массивом:

Visual Basic .NET

```
Dim myStrings(3) As String
myStrings(0) = "A"
myStrings(1) = "String"
myStrings(2) = "Array"
TextBox1.DataBindings.Add("Text", myStrings, "")
```

Visual C#

```
String[] myStrings = new String[3];
myStrings[0] = "A";
myStrings[1] = "String";
myStrings[2] = "Array";
TextBox1.DataBindings.Add("Text", myStrings, "");
```

Чтобы удалить связь с источником данных из элемента управления, вызовите метод *DataBindings.Remove*. Этот метод принимает в качестве параметра объект *Binding*, который получают из свойства *DataBindings*. Следующий пример показывает, как удалить сведения об источнике данных, связанном со свойством *Text* объекта *Label*:

Visual Basic .NET

```
Label1.DataBindings.Remove(Label1.DataBindings("Text"))
```

Visual C#

```
Label1.DataBindings.Remove(Label1.DataBindings["Text"]);
```

Вызвав метод *DataBindings.Clear*, вы удалите все связи с источниками данных у элемента управления:

Visual Basic .NET

```
Label1.DataBindings.Clear()
```

Visual C#

```
Label1.DataBindings.Clear();
```

Управление потоком данных

Управление просмотром записей и обновлением элементов управления, связанных с данными, осуществляется на уровне источника данных при помощи объекта *CurrencyManager*.

Примечание В этом случае источник данных ссылается на одно- или двухмерное хранилище данных, такое, как объект *DataTable*, *DataView*, массив или набор. Объект *DataSet*, содержащий несколько объектов *DataTable*, способен поддерживать несколько источников данных.

Объект *CurrencyManager* «следит» за положением текущей записи в источнике данных. Приложение может использовать несколько источников данных одновременно, при этом каждый из них поддерживает собственный объект *Currency Manager*. Поскольку форма способна отображать содержимое нескольких источников данных одновременно, любая форма управляет связанными с ними объектами *CurrencyManager* посредством главного объекта — *BindingContext*. Этот объект упорядочивает объекты *Currency Manager*, связанные со всеми источниками данных, и предоставляет к ним доступ. Так, свойство *BindingContext* формы позволяет управлять положением текущей записи в любом источнике данных. Чтобы получить доступ к объекту *CurrencyManager* для некоторого источника данных, вызовите свойство *BindingContext*, передав ссылку на объект этого источника данных, например:

Visual Basic .NET

```
Me.BindingContext(DataSet1.Customers)
```

Visual C#

```
this.BindingContext[DataSet1.Customers]
```

Для перемещения между записями устанавливают свойство *Position* соответствующего объекта *BindingContext,* как показано ниже:

Visual Basic .NET

```
' Этот пример предполагает наличие таблицы Customers в
' объекте Dataset с именем DataSet1, содержащимся в текущей форме.
' Сделать первую запись источника данных текущей.
Me.BindingContext(DataSet1.Customers).Position = 0
' Переместить текущую записи на одну позицию вперед,
Me.BindingContext(DataSet1.Customers).Position += 1
' Переместить текущую запись на одну позицию назад,
Me.BindingContext(DataSet1.Customers).Position -= 1
' Сделать пятую запись источника данных текущей.
Me.BindingContext(DataSet1.Customers).Position = 4
' Перейти к последней записи.
Me.BindingContext(DataSet1.Customers).Position - _
 DataSet1.Tables("Customers").Rows.Count - 1
```

Visual C#

```
// Этот пример предполагает наличие таблицы Customers в
// объекте Dataset с именем DataSetl, содержащимся в текущей форме.
// Сделать первую запись источника данных текущей.
this.BindingContext[DataSet1.Customers].Position = 0;
// Переместить текущую записи на одну позицию вперед,
 this.BindingContext[DataSet1.Customers].Position ++;
// Переместить текущую запись на одну позицию назад,
this.BindingContext[DataSet1.Customers].Position --;
// Сделать пятую запись источника данных текущей.
this.BindingContext[DataSet1.Customers].Position - 4;
// Перейти к последней записи.
this.BindingContext[DataSet1.Customers].Position =
   DataSet1.Tables["Customers"].Rows.Count - 1;
```

▶ **Перемещение по записям источника данных, связанного с формой**

Чтобы перейти к заданной записи, установите свойство *Position* соответствующего члена объекта *BindingContext.*

Поскольку в .NET Framework значение свойства *Position* не может быть меньше нуля или больше верхней границы набора, при попытке выбрать запись за пределами источника данных ошибка не возникает — такая команда попросту игнорируется. Тем не менее следует добавить к приложению кол, уведомляющий пользователей (возможно, посредством сообщений) о достижении границ источника данных. Следующий пример демонстрирует применение событий *PositionChanged* объекта *CurrencyManager* для деактивации кнопок перехода между записями по достижении границ источника данных. Как и в предыдущем примере, доступ к объекту *CurrencyManager* здесь осуществляется через свойство *BindingContext* объекта *Form.*

Visual Basic .NET

```
' Объявить обработчик события PositionChanged.
Public Sub OnPositionChanged(ByVal sender As Object, ByVal e As _
   System.EventArgs)
   ' Проверить, не установлен ли CurrencyManager на первую запись.
   If Me.BindingContext(DataSet1, "Customers").Position = 0 Then
      ' Если да, деактивировать кнопку перехода на одну запись назад,
      BackButton.Enabled = False
   Else
      ' в противном случае - активировать эту кнопку.
      BackButton.Enabled = True
   End If
   ' Проверить, не установлен ли объект CurrencyManager на последнюю запись.
   If Me.BindingContext(DataSet1.Customers).Position = _
      DataSet1.Tables("Customers").Rows.Count -1 Then
      ' Если да, деактивировать кнопку перехода на одну запись вперед,
      ForwardButton.Enabled = False
   Else
      ' в противном случае - активировать эту кнопку.
      ForwardButton.Enabled = True
   End If
End Sub

' Не забудьте подключить событие к его обработчику. Для этого
' поместите в конструктор объекта Form следующую строку.
AddHandler Me.BindingContext(DataSet1.Customers).PositionChanged, _
   AddressOf Me.OnPositionChanged
```

Visual C#

```
// Объявить обработчик события PositionChanged.
public void OnPositionChanged(object sender, System.EventArgs e)
{
   // Проверить, не установлен ли CurrencyManager на первую запись.
   if (this.BindingContext[DataSet1.Customers].Position == 0)
      // Если да, деактивировать кнопку перехода на одну запись назад,
      BackButton.Enabled = false;
   else
      // в противном случае - активировать эту кнопку.
      BackButton.Enabled = true;
   // Проверить, не установлен ли объект CurrencyManager на последнюю запись.
   if (this.BindingContext[DataSet1.Customers].Position ==
      DataSet1.Tables["Customers"].Rows.Count -1)
      // Если да, деактивировать кнопку перехода на одну запись вперед.
      ForwardButton.Enabled = false;
   else
      // в противном случае - активировать эту кнопку.
```

```
        ForwardButton.Enabled = true;
}

// Не забудьте подключить событие к его обработчику. Для этого
// поместите в конструктор объекта Form следующую строку,
this.BindingContext[DataSet1.Customers].PositionChanged += new
    EventHandler(this.OnPositionChanged);
```

Сложное связывание данных

Ряд элементов управления, таких, как *ListBox*, *ComboBox* или *DataGrid*, разрешается связать с несколькими записями одновременно, Такое связывание называется *сложным*, а элементы управления, использующие сложное связывание, обычно применяют для отображения нескольких доступных для выбора записей. Чтобы создать элемент управления, использующий сложное связывание, установите свойство *DataSource* элемента управления, поддерживающего такой тип связывания:

Visual Basic .NET

```
DataGrid1.DataSource = DataSet1.Customers
```

Visual C#

```
DataGrid1.DataSource = DataSet1.Customers;
```

В период разработки это можно сделать, установив значение свойства *DataSource* в окне Properties.

Такие элементы управления, как *ListBox*, *Checked List Box* и *ComboBox*, способны отображать несколько записей одновременно, их можно привязать не более чем к одному полю. Для этих элементов управления необходимо записать в свойство *Display Member* значение типа *String*, представляющее имя поля, с которым нужно связать это свойство, например:

Visual Basic .NET

```
ComboBox1.DataSource = DataSet1.Customers
ComboBox1.DisplayMember - "CustomerID"
```

Visual C#

```
ComboBox1.DataSource = DataSet1.Customers;
ComboBox1.DisplayMember = "CustomerlD";
```

Фильтрация и сортировка данных

После заполнения объекта *DataSet* обычно работают с подмножеством его данных, загруженных в память; такое подмножество выделяют при помощи объекта *DataView*. Можно сказать, что этот объект выступает в роли фильтра для объекта *DataTable*, выбирающего из объекта *DataTable* некоторые данные и предоставляющего их элементам управления, связанным с этим объектом *DataTable*. Объект *DataView* поддерживает методы для сортировки и фильтрации данных, а также позволяет обновлять представляемый им объект *DataTable*.

Создание объектов DataView

Чтобы создать объект *DataView,* необходимо передать ссылку на объект *DataTable,* для которого он будет выполнять фильтрацию содержимого, например:

Visual Basic .NET

```
Dim myDataView As New DataView(myDataTable)
```

Visual C#

```
DataView myDataView = new DataView(myDataTable);
```

Показанный код создает объект *DataView,* представляющий данные объекта *myDataTable.* Чтобы задать критерии фильтрации и/или упорядочения данных, следует установить соответствующие свойства объекта *DataView.* Можно создать объект *DataView,* не связанный с объектом *DataTable,* но в этом случае *DataView* остается недоступен для связывания, пока вы не установите его свойство *Table.* Это делается так:

Visual Basic .NET

```
Dim myDataView As New DataView()
myDataView.Table - myDataTable
```

Visual C#

```
DataView myDataView = new DataView();
myDataView.Table = myDataTable;
```

Создать объект *DataView* и связать его с элементами управления допустимо и в период разработки. Для этого перетащите объект *DataView* с вкладки Data панели Toolbox в окно дизайнера и установите значение его свойства *Table* через окно Properties. Далее, чтобы связать созданный объект *DataView,* достаточно установить свойство *DataBindings* (также через окно Properties).

Фильтрация и сортировка содержимого DataSet

Объекты *DataView* позволяют выполнять фильтрацию и сортировку данных, которые они представляют, при этом критерии фильтрации и сортировки разрешается изменять в период выполнения. При изменении критериев фильтрации или упорядочения объекта *DataView* обновляются все связанные с ним элементы управления.

Чтобы упорядочить данные, установите свойство *Sort,* записав в него строку, интерпретируемую как определение правил сортировки данных. Зачастую такая строка содержит имя поля, по которому выполняется сортировка. Вот пример:

Visual Basic .NET

```
myDataView.Sort = "CustomerID"
```

Visual C#

```
myDataView.Sort = "CustomerID";
```

Чтобы задать упорядочение по нескольким полям, разделите их имена запятыми, например:

Visual Basic .NET

```
myDataView.Sort = "State. City"
```

Visual C#

```
myDataView.Sort = "State, City";
```

По умолчанию строки упорядочиваются по возрастанию. Чтобы упорядочить поле по убыванию, добавьте к нему ключевое слово DESC:

Visual Basic .NET

```
' В этом примере строки упорядочиваются, прежде всего, по убыванию
' значения поля State. Далее выполняется сортировка по возрастанию
' значения поля City, поскольку это задано по умолчанию.
myDataView.Sort = "State DESC, City"
```

Visual C#

```
// В этом примере строки упорядочиваются, прежде всего, по убыванию
// значения поля State. Далее выполняется сортировка по возрастанию
// значения поля City, поскольку это задано по умолчанию.
myDataView.Sort = "State DESC, City";
```

Чтобы задать критерий фильтрации данных, запишите в свойство *RowFilter* строку, интерпретируемую как выражение, которое определяет подмножество записей. Например, можно выбрать только строки с заданным значением в некотором поле:

Visual Basic .NET

```
myDataView.RowFilter = "City - 'Seattle'"
```

Visual C#

```
myDataView.RowFilter = "City = 'Seattle'";
```

Выражения, записываемые в *RowFilter*, должны подчиняться синтаксису SQL: строковые литералы необходимо заключать в одинарные кавычки, а даты — между символов «#».

Для более сложных выражений применяют логические операторы AND, OR и NOT:

Visual Basic .NET

```
myDataView.RowFilter = "City = 'Seattle' AND State = 'WA'"
myDataView.RowFilter = "City = 'Seattle' OR State = 'WA'"
myDataView.RowFilter - "City = 'Des Moines' AND (NOT State = 'IA')"
```

Visual C#

```
myDataView.RowFilter - "City = 'Seattle' AND State = 'WA'";
myDataView.RowFilter = "City = 'Seattle' OR State = 'WA'";
myDataView.RowFilter = "City = 'Des Moines' AND (NOT State = 'IA')";
```

Допускаются также арифметические операторы, конкатенация и операторы сравнения:

Visual Basic .NET

```
myDataView.RowFilter = "Length >= 10 AND Height < 4"
myDataView.RowFilter = "CityState = City + 'WA'"
myDataView.RowFilter = "Price * 1.086 <= 500"
```

Visual C#

```
myDataView.RowFilter = "Length >= 10 AND Height < 4";
myDataView.RowFilter = "CityState = City + 'WA'";
myDataView.RowFilter = "Price * 1.086 <= 500";
```

Операторы IN и LIKE служат для поиска заданных строк:

Visual Basic .NET

```
myDataView.RowFilter = "City IN ('Seattle', 'Tacoma', Blaine')"
' Символ "*" является знаком подстановки при сравнении строк. Он заменяет
' любой одиночный символ, а "%" заменяет любую группу символов.
myDataView.RowFilter = "City LIKE 'Se*t%e'"
```

Visual C#

```
myDataView.RowFilter = "City IN('Seattle', 'Tacoma', Blaine')";
// Символ "*" является знаком подстановки при сравнении строк. Он заменяет
// любой одиночный символ, а "%" заменяет любую группу символов.
myDataView.RowFilter = "City LIKE 'Se*t%e'";
```

Свойство *DataView.RowState* также позволяет выполнять фильтрацию объектов *DataRow* на основе их состояния; возможные значения этого свойства перечислены в таблице 6-5.

Таблица 6-5. Значения свойства *RowState*

Значение	Описание
Unchanged	Отображает немодифицированные строки
Added	Отображает строки, добавленные со времени последнего обновления *DataSet*
Deleted	Отображает строки, удаленные со времени последнего обновления *DataSet*
OriginalRows	Отображает исходные версии строк, включая немодифицированные и удаленные
CurrentRows	Отображает все текущие строки, включая добавленные, измененные и немодифицированные
ModifiedCurrent	Отображает текущую модифицированную версию исходных данных
ModifiedOriginal	Отображает исходную версию (если данные модифицированы, является эквивалентом *ModifiedCurrent*)

У этого свойства может быть несколько значений одновременно, Например, установив его в *Added* и *Deleted*, вы получите только добавленные и удаленные строки.

Модификация данных через объект DataView

Объект *DataView* поддерживает три свойства, разрешающие или запрещающие модификацию данных, которые он представляет (таблица 6-6).

Таблица 6-6. Свойства класса *DataView*, связанные с модификацией данных

Свойство	Описание
AllowDelete	Если это свойство установлено в *True* (*true*), *Data View* позволяет удалять строки из представляемого им объекта *DataTable*
AllowEdit	Если это свойство установлено в *True (true),* посредством *DataView* вы сможете модифицировать строки представленного им объекта *DataTable*
AllowNew	Если это свойство установлено в *True (true),* объект *DataView* позволяет добавлять строки к представленному им объекту *DataTable*

По умолчанию *DataView* позволяет произвольно модифицировать объект *DataTable,* который он представляет. Чтобы получить объект *DataView*, доступный только для чтения, либо запретить удаление или вставку строк, установите соответствующие свойства *в false.*

Класс DataViewManager

Класс *DataViewManager* функционирует совместно с классом *DataSet* подобно тому, как класс *Data View* работает в паре с классом *DataTable.* Экземпляр класса *DataViewManager,* связанный с объектом *DataSet*, создает объекты *Data View* для представления различных таблиц этого объекта *DataSet* и управляет ими. При создании нового объекта *DataViewManager* следует передать в качестве параметра объект *DataSet* или установить свойство *DataSet* после создания объекта *DataViewManager,* например:

Visual Basic .NET

```
Dim myDataViewManager As New DataViewManager(myDataSet)
Dim myOtherDataViewManager As New DataViewManager
myOtherDataViewManager.DataSet = myOtherDataSet
```

Visual C#

```
DataViewManager myDataViewManager - new DataViewManager(myDataSet);
DataViewManager myOtherDataViewManager = new DataViewManager();
myOtherDataViewManager.DataSet = myOtherDataSet;
```

Подключившись к объекту *DataSet,* вы получаете возможность управлять его свойствами *RowFilter*, *Sort* и др. с помощью набора *DataView Settings.* В свойстве *DataViewSettings* содержится набор значений свойства *DataView y* каждой из таблиц, составляющих объект *DataSet.* Чтобы установить это значение, укажите таблицу, для которой хотите установить свойство *DataView, а* затем собственно значение свойства. Например, следующий пример устанавливает свойство *RowFilter* объекта *DataView*, связанного с таблицей Customers:

Visual Basic .NET

```
myDataViewManager.DataViewSettings("Customers").RowFilter = _
    "State = 'WA'"
```

Visual C#

```
myDataViewManager.DataViewSettings["Customers"].RowFilter =
    "State = 'WA'";
```

Объект *DataView* можно извлечь из *DataViewManager*, вызвав метод *CreateDataView*; этот метод требует ссылку на объект *DataTable*:

Visual Basic .NET

```
Dim myDataView As DataView
myDataView = myDataViewManager.CreateDataView(DataSet1.Tables(0))
```

Visual C#

```
DataView myDataView;
myDataView = myDataViewManager.CreateDataView(DataSet1.Tables[0]);
```

Резюме

- Связыванием данных называется создание связи между объектами, выступающими в роли провайдера данных и их потребителя. Провайдеры открывают доступ к данным, а потребители получают эти данные и отображают либо обрабатывают их.
- Существует два вида связывания данных — простое и сложное. С элементом управления, использующим первый вид, связано не более одной записи одновременно, тогда как сложное связывание позволяет одновременно связать с объектом все доступные записи.
- Управление связыванием данных с элементами управления осуществляется через свойство *DataBindings*. Любое свойство, доступное в период выполнения, можно связать с источником данных.
- Управление потоком данных осуществляется через объекты *Currency Manager*. С каждым источником данных связан объект *Currency Manager*, отслеживающий позицию текущей записи. Объектами *CurrencyManager* управляют посредством свойства *Form.BindingContext*.
- *DataView* — это объект, связанный с объектом *DataTable* и предоставляющий доступ к упорядоченному и отфильтрованному подмножеству данных базовой таблицы.
- Свойства *RowFilter* и *Sort* объектов *DataView* позволяют задать критерии фильтрации и упорядочения для объекта *DataView*.
- Класс *DataViewManager* управляет всеми объектами *DataView*, связанными с таблицами объекта *DataSet*. Отдельные свойства объекта *DataView* можно установить через свойство *DataViewSettings*; метод *CreateDataView* позволяет получить заданный объект *DataView*.

Занятие 5. Применение XML в ADO.NET

Внутренние механизмы ADO.NET основаны на XML. В этом формате осуществляется представление данных, а также объектов *DataSet* и *DataTable* в памяти; кроме того, поддерживается их запись в текстовые XML-файлы или потоки данных XML. Класс *XmlDataDocument* позволяет напрямую работать с представлениями XML -данных в памяти и синхронизировать их с содержимым объекта *DataSet*. Это занятие посвящено методам ADO.NET, основанным на XML.

Изучив материал этого занятия, вы сможете:

- ✓ получить класс *XmlReader* из объекта *SqlCommand;*
- ✓ заполнить объект *DataSet* данными из текстового файла в формате XML;
- ✓ записать содержимое объекта *DataSet в* XML-поток;
- ✓ записать схему объекта *DataSet* в XML-поток;
- ✓ прочитать XML-схему в объект *DataSet*;
- ✓ создать объект *Xml Data Document* и синхронизировать его данные с объектом *DataSet;*
- ✓ выполнить XSL-преобразование объекта *Xml Data Document.*

Продолжительность **занятия - около** 45 минут.

Извлечение данных XML из баз данных SQL Server 2000

SQL Server 2000 содержит встроенную поддержку доступа к данным XML, благодаря которой удается извлекать из БД и записывать в память данные в этом формате. Эту поддержку обеспечивает провайдер данных ADO.NET для SQL Server.

Получение объекта XmlReader с помощью SqlCommand

Класс *SqlCommand* поддерживает метод для извлечения данных в формате XML — *ExecuteXmlReader*. Этот метод возвращает объект *XmlReader,* предоставляющий данные, которые вернул класс *SqlCommand*, в виде набора строк в формате XML. При этом в SQL-запросе SELECT, который исполняет метод *ExecuteXmi Reader*, должна быть конструкция *FOR XML*. Это метод поддерживает только класс *SqlCommand* и применяется для подключения к БД SQL Server 2000 и выше.

Класс *XmlReader* во многом напоминает *DataReader*: он предоставляет доступ к XML-данным, полученным в результате исполнения запроса, только для однонаправленного чтения. Как и *DataReader,* класс *XmlReader* требуетмонопольного доступа к объекту соединения.

Объект *XmlReader* поддерживает метод *Read,* похожий на метод *Read* объекта *DataReader*, *и* позволяет перебирать узлы XML-документа. Этот метод *Read,* подобно одноименному методу объекта *DataReader,* устанавливает указатель на следующий узел XML-потока и возвращает значение *false* по достижении последнего узла. Как и в случае *DataReader*, следует вызвать метод *Read,* чтобы сделать доступным первый узел. Следующий пример демонстрирует извлечение объекта *XmlReader* и чтение возвращаемого им потока XML:

Visual Basic .NET

```
' Этот пример предполагает наличие действительного
' объекта SqlConnection с именем SqlConnection1.
Dim myReader As Xml.XmlReader
Dim mySQLCommand As New SqlClient.SqlCommand( _
   "SELECT * FROM Customers FOR XML AUTO, XMLDATA", SqlConnection1)
SqlConnection1.Open
myReader = mySQLCommand.ExecuteXmlReader()
While myReader.Read()
   ' Вывести содержимое данного узла и любых дочерних
   ' узлов, включая разметку, на консоль.
   Console.WriteLine(myReader.ReadOuterXml())
End While
myReader.Close
SqlConnection1.Close
```

Visual C#

```
// Этот пример предполагает наличие действительного
// объекта SqlConnection с именем SqlConnection1.
System.Xml.XmlReader myReader;
SqlCommand mySQLCommand = new SqlCommand(
   "SELECT * FROM Customers FOR XML AUTO, XMLDATA", SqlConnection1);
SqlConnection1.Open();
myReader = mySQLCommand.ExecuteXmlReader();
while (myReader.Read())
{
   // Вывести содержимое данного узла и любых дочерних
   // узлов, включая разметку, на консоль.
   Console.WriteLine(myReader.ReadOuterXml());
}
myReader.Close();
SqlConnection1.Close();
```

Применение XML с объектами DataSet

Объект *DataSet* предоставляет методы для взаимодействия с данными, сохраненными в формате XML. Можно загрузить данные, хранящиеся в XML-файле или потоке, в объект *DataSet, а* также записать данные объекта *DataSet* в XML-файл или поток. Разрешается также создавать типизированные объекты *DataSet* с заданной структурой на основе XML-схемы, а также шаблоны для типизированных объектов *DataSet* записью структуры объекта *DataSet в* XML-схему.

Чтение XML-данных в объект DataSet

Метод *DataSet.ReadXml*позволяет получать доступ к хранилищам данных XML. Этот метод принимает существующий XML-файл, поток, объект *XmlReader* или *TextReader*, после чего читает схему и данные из указанного источника в объект *DataSet.*

Например, таким образом удается прочитать XML-данные из файла myData.xml в новый объект *DataSet*

Visual Basic .NET

```
Dim myDataSet As New DataSet
myDataSet.ReadXml("C:\myData.XML")
```

Visual C#

```
DataSet myDataSet = new DataSet();
myDataSet.ReadXml("C:\\myData.XML");
```

Чтобы создать объект *DataSet* с заданной структурой, не загружая в него данные, прочитайте XML-схему с помощью метода *DataSet.ReadXmlSchema*. Подобно методу *ReadXml*, *ReadXmlSchema* принимает существующий XML-файл, поток, объект *XmlReader* или *TextReader*, но записывает в объект *DataSet* только структуру данных, а не их значения. Так, например, можно прочитать XML-схему из файла mySchema.xml:

Visual Basic .NET

```
Dim myDataSet As New DataSet
myDataSet.ReadXmlSchema("C:\mySchema.XML")
```

Visual C#

```
DataSet myDataSet - new DataSet();
myDataSet.ReadXmlSchema("C:\\mySchema.XML");
```

▶ **Как прочитать данные в формате XML в объект *DataSet***

Вызовите метод *DataSet.ReadXml*.

▶ **Как прочитать XML-схему в объект *DataSet***

Вызовите метод *DataSet.ReadXmlSchema*.

Запись содержимого объекта DataSet в формате XML

Объекты *DataSet* способны записывать данные или их схему в XML-файлы. Объект *DataSet* поддерживает метод *WriteXml*, выполняющий экспорт данных в формате XML. Это метод принимает файл, поток, объект *XmlWriter* или *TextWriter* и записывает данные в формате XML из объекта *DataSet* в указанный приемник. Следующий пример демонстрирует запись содержимого объекта *DataSet* в XML-файл myXml.xml; если этот файл не существует, он создается автоматически.

Visual Basic .NET

```
myDataSet.WriteXml("Cz\myData.XML")
```

Visual C#

```
myDataSet.WriteXml("C:\\myData.XML");
```

Аналогичным образом метод *WriteXmlSchema* позволяет записать структуру объекта *DataSet*, не записывая значения данных. Например, так можно записать схему объекта *DataSet* в файл mySchema.xml:

Visual Basic .NET

```
myDataSet.WriteXmlSchema("C:\mySchema.XML")
```

Visual C#

```
myDataSet.WriteXmlSchema("C:\\mySchema.XML");
```

▶ **Как записать содержимое объекта *DataSet* в формате XML**
Вызовите метод *DataSet. WriteXml.*

▶ **Как записать схему объекта *DataSet* в формате XML**
Вызовите метод *DataSet. WriteXmlSchema.*

Применение класса XmlDataDocument

Конструкция класса *XmlDataDocument* позволяет ему работать с объектом *DataSet* напрямую. Класс *XmlDataDocument* является представлением XML-данных в памяти подобно тому, как объект *DataSet* представляет в памяти обычные данные. Управление данными, загруженными в *XmlDataDocument,* построено на основе объектной модели документа W3C Document Object Model (DOM), а объект *XmlDataDocument* способен выполнять роль источника данных для преобразований XSL.

Примечание Манипулирование документами XML не входит в тему этого учебного курса, кроме того, подробный рассказ занял бы несколько глав, поэтому мы ограничимся основами применения класса *XmlDataDocument.*

С каждым объектом *XmlDataDocument* связан объект *DataSet.* При создании объекта *XmlDataDocument* следует передать конструктору ссылку на существующий объект *DataSet* как параметр:

Visual Basic .NET

```
Dim myDocument As New Xml.XmlDataDocument(myDataSet)
```

Visual C#

```
XmlDataDocument myDocument = new XmlDataDocument(myDataSet);
```

В результате создается объект *XmlDataDocument,* синхронизированный с заданным объектом *DataSet.* При этом данные и схема *DataSet* автоматически считываются в *XmlDataDocument.* При внесении любых изменений в один объект, другой объект обновляется автоматически.

▶ **Как создать объект *XmlDataDocument* из существующего объекта *DataSet***

Передайте конструктору класса *XmlDataDocument* ссылку на существующий объект *DataSet* — его данные и схема будут загружены в новый объект *XmlDataDocument.*

Объект *XmlDataDocument* можно также создать, не указывая существующий объект *DataSet.*

Visual Basic .NET

```
Dim myDocument As New Xml.XmlDataDocument()
```

Visual C#

```
XmlDataDocument myDocument - new XmlDataDocument();
```

В этом случае создается пустой объект *DataSet* и связывается с объектом *XmlDataDocument,* В обоих случаях *DataSet*, связанный с заданным объектом *XmlDataDocument,* доступен через свойство *Xml Data Document, DataSet.*

Объект *Xml Data Document* можно заполнить данными в формате XML, взятыми из XML-файла или потока, объекта *XmlReader* или XML-документа, прочитанного при помощи *TextReader*. Чтобы загрузить XML-данные в объект *XmlDataDocument* и синхронизировать их с заданным объектом *DataSet*, прежде всего необходимо вызвать метод *ReadXmlSchema* объекта *DataSet,* указав источник XML-схемы. Заданная схема должна совпадать со структурой источника данных XML; обычно данные и схему извлекают из одного и того же источника XML. Далее следует вызвать метод *Load* объекта *Xml Data Document*, чтобы загрузить XML-данные в память. Вот как загрузить XML-данные из файла myXml.xml в объект *Xml Data Document.*

Visual Basic .NET

```
Dim myDocurnent As New Xml.XmlDataDocument()
myDocument.DataSet.ReadXmlSchema("C:\myXml.xml")
myDocument.Load("C:\myXml.xml")
```

Visual C#

```
XmlDataDocument myDocurnent = new XmlDataDocument();
myDocument.DataSet.ReadXmlSchema("C:\\myXml.xml");
myDocument.Load("C:\\myXml.xml");
```

▶ **Как заполнить объект *XmlDataDocument* данными из источника XML**

1. Создайте экземпляр класса *XmlDataDocument* при помощи конструктора без параметров:

   Visual Basic .NET

   ```
   Dim myDocument As New Xml.XmlDataDocument()
   ```

   Visual C#

   ```
   XmlDataDocument myDocument = new XmlDataDocument();
   ```

2. Вызовите метод *ReadXmlSchema* связанного объекта *DataSet,* чтобы загрузить в него XML-схему:

   Visual Basic .NET

   ```
   myDocument.DataSet.ReadXmlSchema("C:\myXml.xml")
   ```

   Visual C#

   ```
   myDocument.DataSet.ReadXmlSchema("C:\\myXml.xml");
   ```

3. Вызовите метод *XmlDataDocument.Load*, чтобы загрузить данные XML:

Visual Basic .NET

```
myDocument.Load("C:\myXml.xml")
```

Visual C#

```
myDocument.Load("C:\\myXml.xml");
```

Преобразования XSL

Язык XSL создан для преобразования XML-данных в другие форматы. Например, XML-файл — в HTML-документ, пригодный для отображения на Web-странице, либо в другой XML-формат, необходимый некоему приложению.

.NET Framework предоставляет класс *XslTransform*, выполняющий преобразования XSL. Чтобы выполнить преобразование, сначала следует загрузить в объект *XslTransform* таблицу стилей — файл с правилами форматирования XML-данных. Для загрузки таблицы стилей служит метод *XslTransform.Load*. Таблицу стилей можно задать с помощью URL, указывающего на XSL-файл с определениями стилей, либо как список произвольного числа классов, представляющих таблицу стилей, загруженную в память. Следующий пример кода объявляет новый объект *XslTransform* и загружает файл таблицы стилей myStyle.xsl:

Visual Basic .NET

```
Dim myTransform as New Xml.Xsl.XslTransform()
myTransform.Load("C:\myStyle.xsl")
```

Visual C#

```
Xml.Xsl.XslTransform myTransform = new Xml.Xsl.XslTransform();
myTransform.Load("C:\myStyle.xsl");
```

Загрузив таблицу стилей, выполните преобразование объекта *XmlData Document*. Метод *XslTransformTransform* требует три параметра. Первый из них — объект, подлежащий преобразованию. Это может быть *XmlDataDocument* или любой другой объект, в котором реализован интерфейс *IXPathNavigable*. Второй параметр — экземпляр класса *System.Xml.Xsl.XsltArgumentList*; он используется для хранения параметров, необходимых таблице стилей; если таблица стилей не требует параметров, можно передать в нем *Nothing* (*null*). Третьим параметром является целевой объект для записи результатов преобразования. Это может быть экземпляр класса *Stream*, *TextWriter* или *XmlWriter*. Следующий пример демонстрирует преобразование объекта *Xml Data Document* с именем *myDocument* и запись результата в текстовый файл. В этом примере не используются дополнительные параметры, поэтому второй параметр равен *Nothing* (*null*).

Visual Basic .NET

```
' Результат преобразования будет передан классу StreamWriter,
' который запишет его в текстовый файл.
Dim myWriter As New System.IO.StreamWriter("myTextFile.txt")
myTransform.Transform(myDocument, Nothing, myWriter)
```

Visual C#

```
// Результат преобразования будет передач классу StreamWriter,
// который запишет его в текстовый файл.
System.IO.StreamWriter myWriter - new
   System.IO.StreamWriter("myTextFile.txt");
myTransform.Transform(myDocument, null, myWriter);
```

▶ **Исполнение преобразования XSL**

1. Создайте экземпляр класса *System.Xml.Xsl.XslTransform*.
2. Загрузите XSL-таблицу стилей с помощью метола *XslTransform.Load*.
3. Вызовите метод *XslTransform.Transform,* чтобы осуществить преобразование объекта *XmlDataDocument*, при этом нужно передать все параметры таблицы стилей в объекте *XsltArgumentList*.

Резюме

- Внутренние механизмы ADO.NET основаны на XML: в этом формате можно представить данные, хранящиеся как в памяти, так и в файлах.
- Чтобы получить экземпляр класса *XmlReader*, следует вызвать объект *SqlCommand*, представляющий команду SELECT с конструкцией *FOR XML*. Класс *XmlReader* предоставляет доступ только для однонаправленного чтения к присоединенным данным, хранящим данные в формате XML.
- Объект *DataSet* способен читать и записывать данные в формате XML. Методы *ReadXml* и *WriteXml* применяются соответственно для загрузки XML-данных в объект *DataSet* и записи содержимого *DataSet* в этом формате. Объекты *DataSei* также могут читать и записывать XML-схему с помощью методов *ReadSchema* и *WriteSchema*.
- *Xml Data Document* — это представление в памяти XML-документа, синхронизированного с объектом *DataSet,* Любые изменения, внесенные в *DataSet*, передаются непосредственно в связанный с ним *Xml Data Document*, и наоборот.
- Преобразования XSL позволяют конвертировать XML-файл из одного формата в другой. Преобразование выполняет объект *XmlData Document* с использованием таблицы стилей XSL, содержащей определение правил преобразования.

Практикум 6. Подключение к базе данных

Выполнив этот практикум, вы научитесь применять в своих приложениях различные типы доступа к данным, читать содержимое БД с помощью объекта *DataReader*, а также заполнять объект *DataSet* с помощью объекта *DataAdapter* и обновлять БД содержимым объекта *DataSet*. Вы узнаете, как создать типизированный объект *DataSet* с помощью инструмента XML Designer, и научитесь пользоваться мастером Data Form для создания формы, отображающей данные. Решение задач этого практикума хранится в папке \Labs\Ch06\Solution на прилагаемом к книге компакт-диске.

Прежде всего

Для выполнения упражнений этого практикума необходимо установить ядро БД Microsoft Access 2000 или более высокой версии.

Продолжительность **практикума** - около 60 минут.

Упражнение 1. Организация доступа к данным и работа с объектом DataReader

Сейчас вы научитесь получать объекты *DataReader* с помощью объекта *DataCommand* для чтения данных непосредственно в приложение. Прежде всего следует создать соединение с БД при помощи окна Server Explorer.

▶ **Создание нового соединения**

1. Запустите Visual Studio .NET. В меню File щелкните New\Project. Укажите шаблон Windows Application для языка, на котором вы пишете, и щелкните OK.
2. В окне Server Explorer щелкните правой кнопкой узел Data Connections и выберите команду Add Connection — откроется диалоговое окно Data Link Properties.
3. На вкладке Provider выберите провайдер Microsoft Jet 4.0 OLE DB Provider — это провайдер данных для Microsoft Access 2000. Щелкните Next, чтобы перейти на вкладку Connection.
4. В поле Select or Enter a Database Name щелкните кнопку (...), чтобы вызвать диалоговое окно Select Access Database.
5. Найдите файл Doughnut Shop.mdb в папке \Labs\Ch06 и выберите его, щелкнув кнопку Open.
6. Щелкните кнопку Test Connection, чтобы проверить соединение, — должно появиться сообщение об успешной проверке.

Примечание Если такого сообщения вы не получили, вернитесь к пункту 5 и убедитесь, что БД выбрана правильно.

7. Щелкните кнопку OK — в окне Server Explorer появится новое соединение.

▶ **Создание и настройка объекта *DataCommand***

1. Перетащите значок БД из окна Server Explorer на форму — к приложению добавится новый объект *OleDbConnection*, представляющий соединение с этой БД.

2. Перетащите с вкладки Data панели Toolbox на форму экземпляр класса *OleDbCommand* — к приложению добавится новый объект *OleDbCommand* с именем *OleDbCommand1* (*oleDbCommand1*).
3. Присвойте свойствам объекта *OleDbCommandl (oleDbCommandl),* перечисленным в таблице 6-7, указанные в ней значения.

Таблица 6-7. Свойства объекта *OleDbCommandl* (*oleDbCommand1*)

Свойство	Значение
Connection	oleDbConnection1
CommandText	SELECT * FROM Customers

Примечание На вопрос системы о повторной генерации параметров щелкните кнопку Yes.

4. Перетащите с вкладки Windows Forms панели Toolbox на форму элементы управления Button и ListBox. Запишите в свойство *Button1.Text* (*button1.Text*) строку «Click to Execute DataReader».
5. В окне дизайнера дважды щелкните объект *Button1* (*button1*), чтобы создать обработчик по умолчанию для события *Button1.Click (button I. Click)* и открыть его в редакторе кода. Добавьте в обработчик следующий код, извлекающий объект *DataReader u* заполняющий элемент управления ListBox:

Visual Basic .NET

```
Dim myReader As Data.OleDb.OleDbDataReader
Dim CustomerString As String
OleDbConnection1.Open()
myReader = OleDbCommand1.ExecuteReader()
While myReader.Read() = True
    ' Извлечь список имен и фамилий из таблицы
    ' Customers и выполнить их контактенацию.
    CustomerString = myReader(1).ToString() & " " & _
       myReader(2).ToString()
    ' Добавить результат в список ListBox.
    ListBox1.Items.Add(CustomerString)
End While
myReader.Close()
OleDbConnection1.Close()
```

Visual C#

```
System.Data.OleDb.OleDbDataReader myReader;
string CustomerString;
oleDbConnection1.Open();
myReader = oleDbCommand1.ExecuteReader();
while (myReader.Read())
{
```

```
        // Извлечь список имен и фамилий из таблицы
        // Customers и выполнить их контактенацию.
        CustomerString - myReader[1].ToString() + " " +
           myReader[2].ToString();
        // Добавить результат в список ListBox.
        listBox1.Items.Add(CustomerString);
    }
    myReader.Close();
    oleDbConnection1.Close();
```

6. Сохраните и протестируйте приложение. По щелчку кнопки элемент управления ListBox должен заполниться именами и фамилиями покупателей, взятыми из таблицы Customers.

Упражнение 2. Извлечение и обновление данных с помощью объектов DataAdapter и DataSet

Объекты *Data Reader* очень быстро извлекают данные, но не позволяют модифицировать содержимое БД. Сейчас вы реализуете доступ к данным для чтения и записи с помощью объектов *DataAdapter* и *DataSet.* Вы должны заполнить объект *DataSet c* помощью объектов *DataAdapter,* связать *DataSet c* элементом управления DataGrid и обновить БД модифицированными данными.

▶ Как реализовать доступ к БД для чтения и записи

1. Перетащите с вкладки Windows Forms панели Toolbox на поверхность формы элемент управления Button и установите его свойство *Text* в «Click here for Exercise *1*».
2. В окне дизайнера дважды щелкните элемент управления *Button2* (*button2*),чтобы создать обработчик по умолчанию для события *Button2.Click* (*Button2.Click*), и добавьте к нему следующий код:

Visual Basic .NET

```
Dim Exercise2 As New Form2()
Exercise2.Show()
```

Visual C#

```
Form2 Exercise2 = new Form2();
Exercise2.Show();
```

3. В меню File выберите команду Add New Item\Windows Form и щелкните Open, чтобы добавить новую форму.
4. В окне Server Explorer раскройте узел, расположенный слева от нового соединения, чтобы отобразить его дочерние узлы. Щелкните узел, расположенный рядом с узлом Tables, чтобы открыть список таблиц БД.
5. Перетащите значки таблиц Customers, Items и Orders из окна Server Explorer на форму — для всех этих таблиц будут созданы объекты *OleDbDataAdapter,* а также объект *OleDbConnection*, представляющий соединение с БД.

6. Перетащите объект *DataSet* с вкладки Data панели Toolbox на форму — откроется диалоговое окно Add Dataset. Выберите в нем Untyped Dataset и щелкните OK.
7. Перетащите с вкладки Windows Forms панели Toolbox на форму две кнопки и DataGrid. Установите для свойств, перечисленных в таблице 6-8, указанные в таблице значения.

Таблица 6-8. Свойства элементов управления, размещенных на форме Form2

Свойство	Значение
Button J. Text (button 1. Text)	Get Data
Button2. Text (button2. Text)	Update Data
DataGrid 1. DataSource (DataGrid1. DataSource)	DataSet 1

8. В окне конструктора дважды щелкните элемент управления *Button1 (button1)*, чтобы вызвать редактор кода с обработчиком по умолчанию для события *Button1.Click (button 1.Click)*. Добавьте к нему следующий код:

Visual Basic .NET

```
' Поочередно вызовите метод Fill для каждого объекта
' DataAdapter и добавьте результаты к объекту DataSet.
OleDbDataAdapter1.Fill(DataSet1)
OleDbDataAdapter2.Fill(DataSet1)
OleDbDataAdapter3.Fill(DataSet1)
```

Visual C#

```
// Поочередно вызовите метод Fill для каждого объекта
// DataAdapter и добавьте результаты к объекту DataSet,
oleDbDataAdapter1.Fill(dataSet1);
oleDbDataAdapter2.Fill(dataSet1);
oleDbDataAdapter3.Fill(dataSet1);
```

9. В окне дизайнера дважды щелкните *Button2(button2)*, чтобы открыть обработчик по умолчанию для события *Click (button2.Click)* в редакторе кода, и добавьте к нему следующий код:

Visual Basic .NET

```
' Поочередно вызовите метод Update для каждого
' объекта DataAdapter и обновите объект DataSet,
OleDbDataAdapter1.Update(DataSet1)
OleDbDataAdapter2.Update(DataSet1)
OleDbDataAdapter3.Update(DataSet1)
```

Visual C#

```
// Поочередно вызовите метод Update для каждого
// объекта DataAdapter и обновите объект DataSet.
oleDbDataAdapter1.Update(dataSet1);
```

```
oleDbDataAdapter2.Update(dataSet1);
oleDbDataAdapter3.Update(dataSet1);
```

10. Сохраните и протестируйте приложение. Открыв первую форму приложения, щелкните кнопку с надписью «Click here for Exercise 2», чтобы открыть созданную в этом форму. По щелчку кнопки с надписью «Get Data» в элемент управления DataGrid будут загружены данные. Чтобы просмотреть содержимое DataGrid, раскройте в верхнем левом углу DataGrid узел со списком таблиц. Для просмотра содержимого таблицы щелкните соответствующий элемент списка. Значения в DataGrid можно модифицировать и, щелкнув кнопку «UpdateDate», перенести изменения в БД.

Примечание Хотя в БД эти таблицы связаны, в объекте *DataSet* никаких связей между ними нет, поэтому следует проявлять осторожность при обновлении БД. В следующем упражнении вы добавите объект *DataRelations* к типизированному объекту *DataSet*.

Упражнение 3. Создание типизированного объекта DataSet с помощью XML Designer

В предыдущих упражнениях вы научились извлекать нетипизированные данные с помощью объектов *DataReaders* и *DataAdapters*. Сейчас вы с помощью XML Designer создадите типизированный объект *DataSet,* который станет шаблоном для данных приложения.

▶ Создание типизированного объекта *DataSet* с помощью XML Designer

1. Выберите в меню Project команду Add New Item — откроется диалоговое окно Add New Item.
2. В окне Add New Item выберите DataSet и дайте файлу имя **dsDoughnut.xsd** — откроется XML Designer.
3. Перетащите из окна Server Explorer в XML Designer таблицу Customers — добавится новая схема, представляющая таблицу Customers,
4. Перетащите в окно XML Designer таблицы Items и Orders — для каждой таблицы создается новый элемент схемы.

Далее к объекту *DataSet* можно добавить объекты *DataRelation*. Это делается в коде в период выполнения либо с помощью XML Designer в период разработки. В следующем разделе вы средствами XML Designer добавите к схеме объекта *DataSet* объекты *DataRelation*.

▶ Как добавить к схеме объекты *DataRelation* с помощью XML Designer

1. Перетащите объект *Relation* с панели Toolbox на схему таблицы Orders в окне XML Designer —откроется диалоговое окно Edit Relation, показанное на рис. *6-7.*
2. В диалоговом окне Edit Relation задайте для свойств, перечисленных в таблице *6-9,* указанные в ней значения.

Таблица 6-9. Свойства объекта *Data Relation*

Свойство	Значение
Parent element	**Orders**
Child element	**Items**
Key Fields	OrderID
Foreign **Key Fields**	**OrderID**

Остальным свойствам оставьте значения по умолчанию и щелкните кнопку OK — к схеме *DataSet* будет добавлен новый объект *DataRelation*.

3. Перетащите на схему таблицы Orders еще один объект *Relation* и установите для него в диалоговом окне Edit Relation значения свойств, указанные в таблице 6-10.

Таблица 6-10. Свойства объекта *Data Relation*

Свойство	Значение
Parent element	**Customers**
Child element	**Orders**
Key Fields	CustomerID
Foreign Key Fields	CustomerI**D**

4. В меню File выберите команду Save All, чтобы сохранить результаты своей работы.

Рис. 6-7. Диалоговое окно Edit Relation

Упражнение 4. Работа с мастером Data Form

В Visual Studio .NET имеется мастер Data Form. Он упрощает создание форм, предназначенных для просмотра и обновления данных. Сейчас вы с помощью мастера Data Form создадите две формы для просмотра и обновления содержимого БД.

▶ Создание форм для отображения данных

1. Выберите в меню Project команду Add New Item — откроется диалоговое окно Add New Item,
2. Выберите мастер Data Form, назовите новую форму **CustomersOrders.vb или CustomersOrders.cs** и щелкните кнопку Open — запустится мастер Data Form.
3. Щелкните кнопку Next.
4. Выберите объект *DataSet*, который вы хотите использовать в качестве источника данных для формы. Можно создать новый объект *DataSet* либо воспользоваться существующим, созданным в предыдущем упражнении. Во втором случае выберите *dsDoughnut* и щелкните кнопку Next.
5. Мастер автоматически генерирует методы для заполнения и обновления формы, если не указано обратное. Установите флажок Update Button, значения остальных двух раскрывающихся списков оставьте без изменений. Щелкните кнопку Next.
6. Выберите таблицы и поля, которые должна отображать данная форма. В списке таблиц Master or Single выберите таблицу Customers, а в списке Detail — таблицу Orders. Заметьте: ниже отображаются списки полей выбранных таблиц, в которых можно задать поля для размещения на форме. Убедитесь, что в каждом списке указаны все поля, и щелкните кнопку Next.
7. На последней странице мастера можно выбрать способ отображения полей. Разрешается отображать значения записи в элементе управления DataGrid либо в наборе отдельных полей. Если задан второй вариант, мастер Data Form позволяет добавить дополнительные элементы управления для перемещения между записями. Укажите отображение записи в наборе отдельных элементов управления. Убедитесь, что выбраны все дополнительные элементы управления, и щелкните кнопку Finish, чтобы завершить создание формы CustomersOrders, предназначенных для отображения данных.
8. Повторите пункты 1—7, выбрав в списке таблицы Master таблицу Orders, а таблицу Items — в качестве таблицы с подробными данными. Назовите новую форму **OrdersItems.vb** или **OrdersItems.cs**.
9. В окне Solution Explorer щелкните правой кнопкой форму Form2 и затем щелкните команду View Designer — форма Form2 откроется в окне дизайнера.
10. С панели Toolbox перетащите на форму две кнопки, назовите их **btnViewCustomers и btnViewOrders** и установите их свойства *text* соответственно в «View Customers» и «View Orders».
11. Дважды щелкните элемент управления *btnViewCustomers*, чтобы открыть обработчик события btnViewCustomers.Click, и поместите в него следующий код:

Visual Basic .NET

```
Dim myCustomersOrders As New CustomersOrders()
myCustomersOrders.Show()
```

Visual C#

```
CustomersOrders myCustomersQrders = new CustomersOrders();
myCustomersOrders.Show();
```

12. Создайте аналогичный метод, отображающий форму OrdersItems по щелчку кнопки btnViewOrders.
13. Сохраните результаты работы и скомпонуйте проект.

► **Как подключить к БД форму для отображения данных**

Мастер Data Form генерирует для приложения «скелет» инфраструктуры доступа к данным и добавляет код для перемещения между записями. Тем не менее вам потребуется самостоятельно заполнить объект *DataSet* данными, чтобы получить возможность просмотра содержимого БД. Сейчас вы добавите к формам адаптеры данных, управляющие заполнением объекта *DataSet* и обновлением БД.

1. В окне Solution Explorer щелкните правой кнопкой форму CustomersOrders и выберите команду View Designer — откроется окно дизайнера.
2. Перетащите таблицу Customers из окна Server Explorer в окно дизайнера — к форме будут добавлены новые объекты *OleDbDataConnection* и *OleDbDataAdapter*.
3. Перетащите из окна Server Explorer в окно дизайнера таблицы Orders и Items. Обратите внимание, что для каждой таблицы добавляется объект *OleDbDataAdapter*, но дополнительные соединения не создаются,
4. Переименуйте добавленные компоненты, как указано в таблице 6-11,

Таблица 6-11. Имена объектов для доступа к данным

Исходное имя	Новое имя
OleDbDataConnection1 (oleDbDataConnection1)	*dcDoughnut*
OleDbDataAdapter1 (oleDbDataAdapter1)	*daCustomers*
OleDbDataAdapter2 (oleDbDataAdapter2)	*daOrders*
OleDbDataAdapter3 (oleDbDataAdapter3)	*daItems*

5. В окне дизайнера дважды щелкните кнопку Load, чтобы открыть обработчик события *btnLoad.Click*, и добавьте в него код, заполняющий объект *DataSet,*

Visual Basic .NET

```
' Объекты-адаптеры данных заполняют набор данных dsDoughnut,
' сгенерированный мастером Data Form.
daCustomers.Fill(objdsDoughnut)
daOrders.Fill(objdsDoughnut)
daItems. Fill(objdsDoughnut)
```

Visual C#

```
// Объекты-адаптеры данных заполняют набор данных dsDoughnut,
// сгенерированный мастером Data Form.
daCustomers.Fill(objdsDoughnut);
daOrders.Fill(objdsDoughnut);
daItems.Fill(objdsDoughnut);
```

6. В окне Solution Explorer щелкните правой кнопкой форму CustomersOrders и выберите команду View Designer — откроется окно дизайнера. Дважды щелкните кнопку Update, чтобы открыть обработчик события *btnUpdate.Click*, и поместите в него следующий фрагмент кода:

Visual Basic .NET

```
' Адаптеры данных обновляют БД данными модифицированными записей,
daCustomers.Update(objdsDoughnut)
daOrders.Update(objdsDoughnut)
daItems.Update(objdsDoughnut)
```

Visual C#

```
// Адаптеры данных обновляют БД данными модифицированными записей.
daCustomers.Update(objdsDoughnut);
daOrders.Update(objdsDoughnut);
daItems.Update(objdsDoughnut);
```

7. Повторите пункты 1—6 для страниц OrdersItems.
8. Сохраните проект и протестируйте его.

Закрепление материала

Приведенные ниже вопросы помогут вам лучше усвоить основные темы данной главы. Если вы не сумеете ответить на вопрос, повторите материал соответствующего занятия. Ответы для самопроверки — в приложении «Вопросы и ответы» в конце книги.

1. Перечислите основные компоненты провайдера данных и назовите их функции.
2. Кратко опишите особенности доступа к данным в ADO.NET с использованием постоянного подключения и отсоединенных данных.
3. Какие значения принимает свойство *CommandType* объектов *SqlCommand* или *OleDbCommand* и что определяют эти значения?
4. Как в ADO.NET выполнить команду DDL, например ALTER или CREATE TABLE?
5. Кратко опишите преимущества и недостатки применения типизированных объектов *DataSet*.
6. Опишите механизм управления потоком данных в форме, содержащей несколько элементов управления, связанных с данными?
7. Расскажите, как с помощью объекта *DataView* выполнить фильтрацию и упорядочить данные.
8. Дайте краткую характеристику объекту *XmlDataDocument*. Как он связан с объектом *DataSet*?

ГЛАВА 7

Создание элементов управления с помощью .NET Framework

Занятие 1. Применение GDI+ 308

Занятие 2. Создание элементов управления 320

Занятие 3. Решение типичных задач, связанных с разработкой элементов управления 330

В этой главе

В этой главе рассказывается о создании элементов управления с использованием .NET Framework. Вы научитесь создавать производные, составные и формируемые элементы управления, а также решать стандартные задачи, связанные с разработкой элементов управления, и работать с интерфейсом **GDI+.**

Прежде всего

Для изучения материалов этой главы необходимо выполнить занятия глав 1--5.

Занятие 1. Применение GDI+

Windows — операционная система с графическим интерфейсом. Пользователи работают с окнами, которые в графической форме представляют данные и функциональность приложений. Чтобы пользователи могли применять богатые возможности Windows, конструкция приложения должна открывать доступ к его функциональности через графический интерфейс. .NET Framework предоставляет множество графических элементов управления и поддерживает разнообразные методы создания программ с графическим интерфейсом. Однако иногда требуется формировать и отображать собственное графическое представление, а также генерировать элементы управления нестандартного вида. Чтобы в полной мере воспользоваться преимуществами Windows, необходимо освоить возможности интерфейса графических устройств (Graphic Device Interface, GDI). В .NET Framework реализована полностью управляемая версия этого интерфейса под названием GD1+.

Изучив материал этого занятия, вы сможете:

- ✓ создавать и использовать объект *Graphics*;
- ✓ применять перья, кисти и цвета;
- ✓ рисовать и закрашивать контуры;
- ✓ рисовать текст;
- ✓ создавать объекты *GraphicsPath* и *Region*.

Продолжительность занятия - около 40 минут.

Термином GDI+ в .NET Framework обозначают управляемую реализацию GDI, которая применяется для отображения графической информации на экране компьютера. Доступ к этому интерфейсу предоставляется через шесть пространств имен, где реализованы различные виды функций GDI+.

Пространства имен System.Drawing

Пространство имен *System. Drawing* предоставляет чрезвычайно широкие возможности. Подробное исследование компонентов этого пространства имен не является целью данного учебного курса, но здесь мы кратко рассмотрим применение содержащихся в нем классов и методов

В таблице 7-1 описаны пространства имен, расположенные в *System.Drawing*, и основные функции содержащихся в них классов.

Таблица 7-1. Пространства имен *System.Drawing*

Имя	Что содержит
System.Drawing	Основные классы, участвующие в отображении графики на экране. Это ключевое пространство имен для программирования графического интерфейса
System.Drawing.Design	Классы, предоставляющие пользовательскому интерфейсу дополнительные возможности периода разработки
System.Drawing.2D	Классы, отображающие сложные графические эффекты
System.Drawing.Imaging	Классы, предоставляющие дополнительные возможности по манипулированию графическими файлами

Таблица 7-1. *(окончание)*

Имя	Что содержит
System. Drawing, Printing	Классы для печати
System.Drawing. Text	Классы, предоставляющие дополнительные возможности по управлению шрифтами

Объект Graphics

Ключевую роль в формировании и отображении графики играет класс *Graphics* из пространства имен *System.Drawing.* Объект *Graphics* представляет область для рисования, расположенную на поверхности графического элемента, — формы, элемента управления или объекта *Image.* Со всеми этими сущностями связан объект *Graphics,* который позволяет рисовать на их поверхности и отображает все графические элементы, размещенные на ней.

Поскольку объект *Graphics* должен быть связан с графическим элементом, его нельзя создать, вызвав конструктор этого класса напрямую. Вместо этого следует вызвать конструктор объекта *Graphics из* графического элемента. Любой класс-потомок *Control* (включая *Form)* поддерживает метод *CreateGraphics.* Последний возвращает ссылку на объект *Graphics,* связанный с данным элементом управления. Следующий пример демонстрирует получение доступа к объекту *Graphics* формы *myForm:*

Visual Basic .NET

```
Dim myGraphics as System. Drawing. Graphics
myGraphics = myForm.CreateGraphics()
```

Visual C#

```
System.Drawing.Graphics myGraphics;
myGraphics = myForm.CreateGraphicsO;
```

В результате исполнения этого кода создается объект *Graphics,* которым можно пользоваться для отображения графики на поверхности формы.

При работе с изображениями метод *Graphics. From Image* позволяет создать объект *Graphics,* связанный с заданным объектом */mage.* Это статический метод, поэтому для его вызова ссылка на объект *Graphics* не требуется. Объектом *Image* считается любой объект-потомок класса *Image,* например *Bitmap.* Вот как создать из файла объект *Bitmap* и получить связанный с ним объект *Graphics:*

Visual Basic .NET

```
Dim myImage As New Bitmap("C:\myImage.bmp")
Dim myGraphics As System.Drawing.Graphics
myGraphics = Graphics.FromImage(myImage)
```

Visual C#

```
Bitmap myImage = new Bitmap("C:\\myImage.bmp");
System.Drawing.Graphics myGraphics;
myGraphics = Graphics.FromImage(myImage);
```

Заметьте, что даже невидимый объект *Image* позволяет создать объект *Graphics* и манипулировать им,

Координаты

Формирование изображения происходит в области, заданной границами элемента управления. Здесь действует двухмерная система координат, каждая точка которой задана парой значений — X и Y. По умолчанию начальная точка системы координат (0,0) каждого элемента находится в его верхнем левом углу, а координаты задаются в пикселах. Пространство имен *System.Drawing* содержит ряд структур (таблица 7-2), описывающих точку или область в данной системе координат.

Таблица 7-2. Структуры, описывающие координаты точек и фигур

Имя	Описание
Point	Представляет точку с координатами *X* и *Y*, заданными значениями типа *Integer (Int)*
PointF	Представляет точку с координатами *X* и *Y*, заданными значениями типа *Single* (*float*)
Size	Представляет прямоугольную область, размеры которой заданы парой значений *Height* и *Width* типа *Integer*
SizeF	Представляет прямоугольную область, размеры которой заданы парой значений *Height* и *Width* типа *Single* (*float*)
Rectangle	Представляет прямоугольную область, границы которой заданы значениями *Top, Bottom, Left* и *Right* типа *Integer (Int)*
RectangleF	Представляет прямоугольную область, границы которой заданы значениями *Top, Bottom, Left* и *Right* типа *Single (float)*

Как видно из таблицы 7-2, существует два типа подобных структур: те, что принимают целочисленные значения, и те, что используют числа с плавающей точкой. Структуры *Point, Size* и *Rectangle* первого типа разрешается явно преобразовывать в соответствующие структуры второго типа. Чтобы выполнить такое преобразование, следует явно преобразовать каждую координату из типа с плавающей точкой в целочисленный тип, например:

Visual Basic .NET

```
Dim myPoint As Point
Dim myPointF As New PointF(13.5,33.21)
myPoint = New Point(CInt(myPointF.X), CInt(myPointF.Y))
```

Visual C#

```
Point myPoint;
PointF myPointF = new PointF(13.5F,33.21F);
myPoint = new Point((int)myPointF.X, (int)myPointF.Y);
```

Обратите внимание на важную особенность структур *Size* и *Rectangle:* обе они определяют прямоугольную часть клиентской области, однако *Size* определяет лишь размер, но не положение прямоугольника, тогда как структура *Rectangle* задает реальное положение прямоугольника в клиентской области. Для создания *Rectangle*

требуются две структуры: *Size* и *Point*, указывающие положение верхнего левого угла прямоугольника:

Visual Basic .NET

```
Dim myOrigin As New Point(10, 10)
Dim mySize As New Size(20, 20}
' Создать прямоугольник размером 20 на 20 пикселов, верхний
' левый угол которого расположен в точке с координатами (10,10).
Dim myRectangle As New Rectangle(myOrigin, mySize)
```

Visual C#

```
Point myOrigin = new Point(10, 10);
Size mySize = new Size(20, 20);
// Создать прямоугольник размером 20 на 20 пикселов, верхний
// левый угол которого расположен в точке с координатами (10,10),
Rectangle myRectangle - new Rectangle(myOrigin, mySize);
```

Рисование фигур

Объект *Graphics* инкапсулирует множество методов для рисования на экране простых и сложных фигур. Существует две основных разновидности этих методов. К первой относятся методы, имена которых начинается с «*Draw*» (таблица 7-3). Эти методы используются для рисования прямых, дуг и контуров фигур. Другая разновидность методов, имена которых начинается с *«Fill»* (таблица 7-4), служит для рисования залитых фигур — прямоугольников, эллипсов и многоугольников.

Эти методы принимают различные комбинации параметров, определяющих размеры и положение отображаемых фигур. Кроме того, им требуется объект, формирующий изображение фигуры, Для контурных фигур это *Pen,* а для закрашенных — *Brush*.

Таблица 7-3. Методы для рисования контурных фигур

Имя	Описание
DrawArc	Рисует дугу, представляющую часть эллипса
DrawBezier	Рисует кривую Безье
DrawBeziers	Рисует несколько кривых Безье
DrawClosedCurve	Проводит через несколько точек замкнутую кривую
DrawCurve	Проводит через несколько точек незамкнутую кривую
DrawEllipse	Рисует эллипс, вписанный в прямоугольник
DrawLine	Соединяет две точки линией
DrawLines	Соединяет линиями несколько точек
DrawPath	Рисует заданный объект *GraphicsPath*, представляющий сложную фигуру
DrawPie	Рисует сектор окружности
DrawPolygon	Рисует многоугольник по заданному числу точек
DrawRectangle	Рисует прямоугольник
DrawRectangles	Рисует несколько прямоугольников

Таблица 7-4. Методы для рисования закрашенных фигур

Имя	Описание
FillClosedCurve	Рисует **замкнутую** кривую, заданную массивом **точек, и** закрашивает **полученную фигуру**
FillEllipse	**Рисует закрашенный** эллипс
*Fill**Path*** Рисует	и **закрашивает** объект *Graphics Path,* представляющий сложную **фигуру**
FillPie	**Рисует закрашенный сектор**
FillPolygon	Рисует **многоугольник, заданный массивом** точек, **и закрашивает его**
FillRectangle	Рисует закрашенный **прямоугольник**
FillRectangles	**Рисует несколько закрашенных прямоугольников**
*Fill**Region***	**Рисует и закрашивает объект *Region,* обычно** представляющий **сложную фигуру**

Объекты Color, Brush и Pen

Объекты *Color. Brush* и *Pen* формируют облик графического элемента. Объект *Brush* применяют для отображения закрашенных фигур, *Pen* — для отображения прямых и дуг, а объект *Color* задает цвет фигуры.

Color

Структура *Color* располагается в пространстве имен *System.Drawing* и представляет некоторый цвет. Каждый цвет определяется четырьмя значениями: *Alpha*, представляющим прозрачность, и тремя значениям *Red, Green* и *Blue* из диапазона 0-255. Чтобы создать новый цвет, необходимо передать эти значения методу *Color.From-Argb*:

Visual Basic .NET

```
Dim myColor As Color
nyColor = Color.FromArgb(128, 255, 12, 43)
```

Visual C#

```
Color myColor;
nyColor = Color.FromArgb(128, 255, 12, 43);
```

Если вы не используете эффект прозрачности, параметр *Alpha* можно пропустить, указав только значения *Red, Green* и *Blue:*

Visual Basic .NET

```
Dim myColor As Color
myColor = Color.FromArgb(255, 12, 43)
```

Visual C#

```
Color myColor;
myColor = Color.FromArgb(255, 12, 43);
```

Кроме того, множество стандартных цветов доступно в .NET Framework в виде именованных цветов:

Visual Basic .NET

```
Dim myColor As Color
myColor = Color.Tomato
```

Visual C#

```
Color myColor;
myColor = Color.Tomato;
```

Brush

Объект *Brush* представляет кисть и служит для рисования закрашенных фигур. Все разновидности кистей являются потомками абстрактного класса *Brush* и позволяют создавать закрашенные фигуры различного вида. Типы кистей и пространства имен, в которых расположены соответствующие объекты, описаны в таблице 7-5.

Таблица 7-5. Типы кистей

Тип	Пространство имен	Описание
SolidBrush	*System.Drawing*	Сплошная одноцветная кисть
TextureBrush	*System.Drawing*	Кисть для заливки замкнутого контура изображением
HatchBrush	*System.Drawing.Drawing2D*	Кисть для узорной заливки
LinearGradientBrush	*System.Drawing.Drawing2D*	Кисть для заливки двухцветным градиентом
PathGradientBrush	*System.Drawing.Drawing2D*	Кисть, использующая сложную градиентную запивку

Для создания объекта *SolidBrush* достаточно указать цвет, например:

Visual Basic .NET

```
Dim myBrush As New SolidBrush(Color.PapayaWhip)
```

Visual C#

```
SolidBrush myBrush = new SolidBrush(Color.PapayaWhip);
```

Конструкторы других кистей сложнее и требуют дополнительных параметров. Например, для создания кисти типа *TextureBrush* необходим объект *Image*, *а* для *LinearGradientBrush* — два цвета и ряд других параметров, в зависимости от типа выбранного конструктора.

Pen

Объекты *Pen* представляют перья, они предназначены для рисования прямых и дуг, а также для применения к контурным фигурам различных графических эффектов. Существует только один класс *Pen,* и он является *запечатанным* (sealed). Создать объект класса *Pen* несложно, достаточно указать его цвет:

Visual Basic .NET

```
Dim myPen As New Pen(Color.BlanchedAlmond)
```

Visual C#

```
Pen myPen = new Pen(Color.BlanchedAlmond);
```

В результате исполнения предыдущего кода создается перо, ширина которого по умолчанию равна 1 пикселу. Ширину задают в конструкторе объекта *Pen* следующим образом;

Visual Basic .NET

```
Dim myPen As New Pen(Color.Lime, 4)
```

Visual C#

```
Pen myPen - new Pen(Color.Lime, 4):
```

Этот код назначает ширину пера, равную четырем пикселам. Перо можно создать и на основе имеющейся кисти, вид таких перьев будет согласован со стилем интерфейса. Это особенно удобно при использовании сложных теней и других эффектов. Следующий пример демонстрирует создание пера на основе имеющейся кисти *myBrush*:

Visual Basic .NET

```
Dim myPen As New Pen(myBrush)
```

Visual C#

```
Pen rnyPen - new Pen(myBrush);
```

При создании пера на основе кисти также разрешается задавать ее ширину.

Системные кисти, перья и цвета

При разработке пользовательского интерфейса иногда нестандартные компоненты приходится оформлять в том же стиле, что и системные. Для этого .NET Framework предоставляет доступ к системным цветам через класс *SystemColors*, который содержит набор статических членов, представляющих текущие системные цвета. Так, при разработке нестандартных компонентов пользовательского интерфейса можно назначить им системные цвета, чтобы в период выполнения они отображались с использованием текущей системной палитры. Получить доступ к системным цветам можно так:

Visual Basic .NET

```
Dim myColor as Color = SystemColors.HighlightText
```

Visual C#

```
Color myColor = SystemColors.HighlightText;
```

Наряду с *SystemColors*, в .NET Framework также имеются классы *SystemPens* и *SystemBrushes*, предоставляющие доступ к перьям и кистям, которые применяются

по умолчанию. Эти кисти работают так же, как любые другие, представленные классами *Pen* или *SolidBrush*.

Рисование простых фигур

Класс *Graphics* поддерживает ряд методов, позволяющих рисовать простые фигуры (таблица 7-3).

Любым методам для рисования контурных фигур необходим действительный объект *Pen*, а методам, рисующим закрашенные фигуры, — действительный объект *Brush*. Кроме того, при вызове этим методам следует передать любые требуемые им объекты. Так, например, можно нарисовать прямоугольник средствами метода *DrawRectangle*:

Visual Basic .NET

```
' Создать объект Rectangle.
Dim myRectangle As New Rectangle(0, 0, 30, 20)
' Создать объект Graphics, связанный с формой.
Dim g As Graphics - Me.CreateGraphics()
' Нарисовать прямоугольник системным пером.
g.DrawRectangle(SystemPens.ControlDark, myRectangle)
' Освободить ресурсы, занятые объектом Graphics.
g.Dispose()
```

Visual C#

```
// Создать объект Rectangle.
Rectangle myRectangle = new Rectangle(0, 0, 30, 20);
// Создать объект Graphics, связанный с формой,
Graphics g = this.CreateGraphics();
// Нарисовать прямоугольник системным пером.
g.DrawRectangle(SystemPens.ControlDark, myRectangle);
// Освободить ресурсы, занятые объектом Graphics.
g.Dispose();
```

Закончив работу с объектом *Graphics*, обязательно вызовите его метод *Dispose*. В противном случае быстродействие приложения может снизиться, так как этот объект использует много системных ресурсов. Аналогично следует поступать с любыми созданными вами объектами *Pen* и *Brush*. Следующий пример кода показывает, как нарисовать закрашенный эллипс и правильно освободить объекты *Brush* и *Graphics*:

Visual Basic .NET

```
Dim myBrush As New SolidBrush(Color.MintCream)
Dim g As Graphics = Me.CreateGraphics()
' Нарисовать эллипс, вписанный в прямоугольник.
Dim myRectangle As New Rectangle(0, 0, 30, 20)
g.FillEllipse(myBrush, myRectangle)
' Освободить ресурсы объектов Graphics и Brush.
g.Dispose()
myBrush. Dispose()
```

Visual C#

```
SolidBrush myBrush = new SolidBrush(Color.MintCream);
Graphics g = this.CreateGraphics();
// Нарисовать эллипс, вписанный в прямоугольник.
Rectangle myRectangle = new Rectangle(0, 0, 30, 20);
g.FillEllipse(myBrush, myRectangle);
// Освободить ресурсы объектов Graphics и Brush.
g.Dispose();
myBrush.Dispose();
```

▶ **Как нарисовать простую фигуру**

1. Создайте объект *Graphics,* представляющий область, в которой вы хотите нарисовать фигуру.
2. Создайте необходимые вспомогательные объекты. К ним относятся объекты, задающие координаты и размеры фигур, например *Point* или *Rectangle*, а также объекты *Pen* (для рисования контурных фигур) и *Brush* (для закрашенных фигур).
3. Вызовите соответствующий метод объекта *Graphics.*
4. Освободите все ресурсы, занятые созданными объектами *Pen* или *Brush.*
5. Освободите ресурсы объекта *Graphics.*

Отображение текста

Метод *DrawString* объекта *Graphics* позволяет отобразить на экране текстовую строку как графический элемент, нарисовав ее в определенном месте экрана заданной кистью. Помимо объекта *Brush*, этому методу необходимо передать координаты верхнего левого угла отображаемого текста (в виде структуры *PointF*). Ниже показан пример кода, отображающего строку с помощью метода *Graphics.DrawString*:

Visual Basic .NET

```
' В этом примере используется одна из
' системных кистей класса SystemBrush.
Dim g As Graphics = me.CreateGraphics()
Dim myString As String = "Hello World"
Dim myFont As New Font("Times New Roman", 36, FontStyle.Regular)
' Последние два параметра представляют координаты строки.
g.DrawString(myString, myFont, SystemBrushes.Highlight, 0, 0)
' Не забудьте освободить объект Graphics!
g.Dispose()
```

Visual C#

```
// В этом примере используется одна из
// системных кистей класса SystemBrush.
Graphics g = this.CreateGraphics();
String myString = "Hello World";
Font myFont = new Font("Times New Roman", 36, FontStyle.Regular);
Последние два параметра представляют координаты строки.
g.DrawString(myString, myFont, SystemBrushes.Highlight, 0, 0);
```

```
// Не забудьте освободить объект Graphics!
g.Dispose();
```

▶ **Отображение текста как графического элемента**

1. При необходимости создайте объекты *Font* и *Brush,* определяющие вид объекта *String.*
2. Получите ссылку на объект *Graphics,* связанный с областью, в которой должен быть отображен текст.
3. Вызовите метод *Graphics.DrawString,* передав объекты *String, Font, Brush* и координаты верхнего левого угла области, в которой будет отображен текст.
4. Освободите объект *Graphics,* вызвав метод *Graphics.Dispose.*

Рисование сложных фигур

Время от времени приходится рисовать фигуры более сложные, чем прямоугольники, эллипсы и многоугольники. У объектов простых фигур есть встроенные методы, облегчающие их рисование, тогда как рисование сложных фигур требует дополнительной подготовки. Ключевым для визуализации сложных фигур является объект *GraphicsPath* из пространства имен *System.Drawing.Drawing2D.* Он описывает произвольный замкнутый контур или набор фигур. Так, можно создать объект *GraphicsPath,* образованный эллипсами, прямоугольниками и другими простыми объектами, представляющий фигуру неправильной формы.

Создание объекта GraphicsPath

Объект *GraphicsPath* создают вызовом одной из версий его конструктора, Простейшая версия конструктора объекта *GraphicsPath* не принимает параметров:

Visual Basic .NET

```
Dim myPath As New Drawing2D.GraphicsPath()
```

Visual C#

```
GraphicsPath myPath = new Drawing2D.GraphicsPath();
```

Кроме того, объект *GraphicsPath* допустимо задавать в виде массива точек и значений типа *Byte.* Точки определяют координаты контура, а значения *Byte* — тип линий, которыми эти точки соединяются. Преобразовав подобный массив из перечислимого типа *System.Drawing.Drawing2D.PathPointType,* вы сможете написать более понятный и удобный в сопровождении код. Вот пример кода, создающего очень простой объект *GraphicsPath:*

Visual Basic .NET

```
' Этот пример предполагает наличие оператора Imports System.Drawing.Drawing2D
Dim myPath As New GraphicsPath(New Point() {New Point(1, 1), _
   New Point(32, 54), New Point(33, 5)}, New Byte() _
   {CType(PathPointType.Start, Byte), CType(PathPointType.Line, _
   Byte), CType(PathPointType.Bezier, Byte)})
```

Visual C#

```
// Этот пример предполагает наличие оператора using System.Drawing.Drawing2D
GraphicsPath myPath = new GraphicsPath(new Point[] {new Point(1, 1),
    new Point(32, 54), new Point(33, 5)}, new byte[] {
    (byte)PathPointType.Start, (byte)PathPointType.Line,
    (byte)PathPointType.Bezier});
```

К созданному объекту *GraphicsPath* можно добавлять фигуры. Фигура — это замкнутый контур, он может быть простым (как эллипс или прямоугольник) или сложным (как произвольные кривые и контуры символов).

Методы класса *GraphicsPath* позволяют добавлять к контуру новые фигуры (таблица 7-6).

Таблица 7-6. Методы для добавления фигур к объекту *GraphicsPath*

Имя	Что добавляет
AddClosedCurve	Замкнутую кривую, заданную массивом точек
AddEllipse	Эллипс
AddPath	Заданный экземпляр объекта *GraphicsPath*
AddPie	Окружность
AddPolygon	Многоугольник, заданный массивом точек
AddRectangle	Прямоугольник
AddRectangles	Массив прямоугольников
AddString	Графическое представление строки символов с заданным начертанием

Разрешается не только напрямую добавлять фигуры к объекту *GraphicsPath*, но и составлять контуры, добавляя прямые, кривые и дуги. Чтобы начать новый контур, вызовите метод *GraphicsPath.StartFigure*. Далее, используя методы класса *GraphicsPath,* можно добавлять к контуру новые фигуры. Закончив конструирование фигуры, вызовите метод *GraphicsPath.CloseFigure*, чтобы замкнуть ее контур — последняя точка фигуры автоматически соединяется с первой. Вот пример:

Visual Basic .NET

```
Dim myPath As New Drawing2D.GraphicsPath()
myPath.StartFigure()
' Здесь должен быть код, добавляющий элементы контура.
myPath.CloseFigure()
```

Visual C#

```
GraphicsPath myPath = new Drawing2D.GraphicsPath();
myPath.StartFigure();
// Здесь должен быть код, добавляющий элементы контура.
myPath.CloseFigure();
```

Если вызвать метод *StartFigure* два раза подряд, не вызвав между ними метод *CloseFigure,* то фигура, созданная первым вызовом *StartFigure,* останется незамкнутой.

Примечание При отображении в период выполнения любая незамкнутая *фигура* автоматически замыкается добавлением прямой, соединяющей первую и последнюю точку контура этой фигуры.

В таблице 7-7 описаны методы объекта *GraphicsPath,* позволяющие добавлять новые элементы контура фигуры.

Table 7-7. Методы для добавления элементов контура

Имя	Что добавляет
AddArc	Дугу
AddBezier	Кривую Безье
AddBeziers	Набор кривых Безье
AddCurve	Кривую, заданную массивом точек
AddLine	Прямую
AddLines	Набор соединенных прямых

▶ **Как нарисовать сложную фигуру**

1. Получите ссылку на объект *Graphics,* связанный с областью рисования, в которой нужно нарисовать фигуру.
2. Создайте новый экземпляр класса *GraphicsPath.*
3. Добавьте к нему фигуры при помощи методов класса *GraphicsPath.*
4. Вызовите метод *Graphics. DrawPath,* чтобы нарисовать контур, или *Graphics. FillPath,* чтобы нарисовать закрашенную фигуру, заданную объектом ***GraphicsPath.***
5. Освободите объект *Graphics.*

Резюме

- *Graphics* — ключевой объект для рисования графики. Он представляет поверхность области рисования и предоставляет методы для рисования на ней.
- Объекты *Pen, Brush* и *Color* управляют отображением графики в области рисования. Объекты *Pen* рисуют линии, *Brush* — заливают фигуры, а объекты *Colors* задают цвета. Применение классов *SystemPens, SystemBrushes* и *System Colors* позволяет согласовать внешний вид компонента и приложения в целом,
- Методы объекта *Graphics* отображают простые фигуры и текст.
- Сложные фигуры можно определить средствами объекта *GraphicsPath.* Он поддерживает ряд методов, которые упрощают рисование сложных фигур. Готовый объект *GraphicsPath* можно визуализировать с помощью объекта *Graphics.*

Занятие 2. Создание элементов управления

В основе программирования с применением Windows Forms лежит использование элементов управления. Они инкапсулируют отдельные функции приложения и являются их графическим представлением для пользователя. .NET Framework предоставляет в распоряжение разработчика ряд инструментов и методов для создания собственных элементов управления. На этом занятии вы познакомитесь с различными типами нестандартных элементов управления и инструментами для их создания.

Изучив материал этого занятия, вы сможете:

- ✓ описать различные типы нестандартных элементов управления;
- ✓ создавать все типы нестандартных элементов управления;
- ✓ добавлять к элементам управления свойства, методы и события;
- ✓ создавать производные элементы управления;
- ✓ создавать составные элементы управления;
- ✓ создавать формируемые элементы управления.

Продолжительность занятия - около 45 минут.

Элементы управления .NET Framework — это специализированные классы, которые содержат код, формирующий их графический интерфейс. Существует несколько источников элементов управления, применяемых в приложениях. Во-первых, это стандартные элементы управления Visual Studio .NET, разработанные Microsoft. Они обеспечивают широкие возможности и позволяют разрабатывать клиентские приложения с богатой функциональностью. Во-вторых, в приложениях допустимо применять элементы управления от сторонних разработчиков, а также существующие элементы управления ActiveX. Элементы управления от сторонних разработчиков поддерживают возможности, отсутствующие у стандартных компонентов из .NET Framework. Наконец, если функции, необходимые вашему приложению, не реализованы ни в стандартных элементах управления .NET Framework, ни в элементах управления сторонних разработчиков, разрешается создавать собственные элементы управления.

Принципы создания элементов управления

Все элементы управления .NET Framework являются прямыми или косвенными потомками класса *Control* и наследуют от него базовый набор низкоуровневой функциональности, необходимой всем элементам управления. Например, *Control* предоставляет код для обработки данных, введенных пользователем с помощью клавиатуры и мыши, а также код, взаимодействующий с ОС. Кроме того, классы элементов управления получают от своего предка набор свойств, методов и событий, общих для всех элементов управления. Однако базовый класс не передает своим потомкам ни специфических функций, ни кода, формирующего интерфейс элемента управления.

Основных подходов к созданию элементов управления три:

- наследование от существующих элементов управления;
- объединение стандартных элементов управления в группы;
- написание нестандартных элементов управления «с нуля».

Наследование от существующих элементов управления

Проще всего создать новый элемент управления, объявив его на основе существующего. Созданный таким образом элемент управления не только сохраняет всю функциональность базового компонента, но и позволяет дополнить ее новыми возможностями.

Помимо функциональности, производный элемент управления унаследует от своего предка вид интерфейса. Например, элемент управления, производный от *Button,* внешне не отличается от обычной кнопки. Это освобождает от создания интерфейса для производных элементов управления, но при необходимости это можно сделать.

Большинство элементов управления Windows Forms, если они не определены с ключевым словом *NotInheritable(sealed),* могут служить базовыми классами для новых элементов управления. Например, это позволяет создать новый элемент управления *TextBox* с встроенным кодом для проверки значения поля или *PictureBox,* поддерживающий установку пользователем фильтров для отображаемых в нем изображений. Наследование также позволяет создавать элементы управления, функционально идентичные базовому элементу управления, но отличающиеся по внешнему виду. Пример — нестандартная кнопка, которая отличается от традиционной прямоугольной кнопки круглой формой.

Объявление новых элементов управления при помощи наследования считается простейшим способом их создания и требует минимальных затрат времени. Этот подход оптимален, если требуется передать большинство или все возможности существующего элемента управления Windows Forms новому элементу управления, добавив лишь несколько нестандартных функций. Данный способ удобен и в тех случаях, когда требуется изменить облик элемента управления, сохранив его функциональность. Если же требуется создать элемент управления с кардинально иной функциональностью, не следует объявлять его на основе существующего; *к* этом случае применяются способы, описанные в следующих разделах.

Наследование от класса UserControl

Иногда функциональности одного элемента управления недостаточно для решения поставленных задач. Предположим, вам требуется элемент управления, поддерживающий связывание с источником данных и отображающий имя, фамилию и номер телефона в элементах управления *TextBox.* Можно добавить к форме соответствующий код, но рациональнее создать новый элемент управления, содержащий несколько текстовых полей. Это особенно удобно, когда подобные возможности требуются нескольким приложениям. Элементы управления, образованные объединенными в группу стандартными элементами управления Windows Forms, называются *пользовательскими* или *составными.*

Составные элементы управления объявляют на основе класса *UserControl.* предоставляющего базовую функциональность, которую можно расширить путем добавления свойств, методов, событий и других элементов управления. Дизайнер составных элементов управления позволяет размещать на их поверхности стандартные элементы управления Windows Forms и добавлять код, реализующий специфические возможности. Таким образом, пользователь может создавать новые элементы управления, объединяя стандартные элементы управления Windows Forms в логическиегруппы.

При разработке графического интерфейса пользовательских элементов управления доступны не все возможности, но разработчику, как правило, достаточно лишь настроить и правильно расположить стандартные элементы управления.

Создание пользовательского элемента управления не всегда простая задача — это зависит от требований к функциональности нового компонента. Способ, описанный в этом разделе, удобен для создания новых компонентов, сочетающих функциональность стандартных элементов управления и нестандартные возможности.

Наследование от класса Control

Если требуется более сложный интерфейс или возможности, которые не удается реализовать с применением составных и производных элементов управления, обращаются к созданию *формируемых* элементов управления. Формируемые элементы являются прямыми потомками *Control* — базового класса всех элементов управления. Класс *Control* передает «по наследству» большой объем функциональности, общей для всех элементов управления, например поддержку взаимодействия с мышью и общие события (например, *Click),* а также облегчающие работу с элементами управления свойства, такие, как *Font, ForeColor, BackColor, Visible* и т.д.

Тем не менее класс *Control* не предоставляет производным от него элементам управления никаких специальных возможностей. Всю логику, необходимую для решения специфичных задач данного элемента управления, должен реализовать разработчик. Хотя класс *Control* предоставляет значительную часть свойств, определяющих облик интерфейса, разработчику придется самостоятельно написать весь код графического интерфейса элемента управления.

Создание формируемых элементов управления требует больше всего времени и усилий от программиста, поскольку на его плечи ложится не только реализация нестандартных возможностей данного элемента управления, но и создание его графического представления, что может отнять массу времени. Формировать элементы управления самостоятельно следует, только если требуется сложная форма либо функциональность, которую не предоставляет ни один другой тип элементов управления.

Добавление членов к элементам управления

Процедура добавления свойств, методов, полей и событий к элементам управления не отличается от добавления членов к обычному классу. Члены классов элементов управления разрешается объявлять с любым подходящим уровнем доступа, чтобы сделать их доступными форме, на которой размещен элемент управления.

Открытые свойства, добавленные к элементам управления в период разработки, автоматически отображаются в окне Properties, где их может отредактировать и настроить любой пользователь. Если вы не хотите, чтобы свойство отображалось в окне Properties, добавьте к его объявлению атрибут *Browsable* и установите его в *false*. Вот как это делается:

Visual **Basic** **.NET**

```
<System.ComponentModel.Browsable(False)> _
   Public Property StockNumber() As Integer
' Реализация свойства опущена.
End Property
```

Visual C#

```
[System.ComponentModel.Browsable(false)]
public int StockNumber
{
   // Реализация свойства опущена.
}
```

Создание производных элементов управления

Производный элемент управления содержит всю функциональность своего базового класса, а также ряд дополнительных возможностей. Чтобы создать производный элемент управления, необходимо указать для него базовый класс, например класс стандартного элемента управления Windows Forms. Новый элемент управления, созданный таким образом, сохранит внешний вид и поведение своего предка. Ниже показан пример создания элемента управления, производного от *Button;*

Visual Basic .NET

```
Public Class myButton
   Inherits System.Windows.Forms.Button
   ' Остальной код опущен.
End Class
```

Visual C#

```
public class myButton : System.Windows.Forms.Button
{
   // Остальной код опущен.
}
```

Реализация новых возможностей в производных элементах управления

Как правило, к производным элементам управления прибегают, когда требуется реализовать в них дополнительные функции, отсутствующие у стандартных элементов управления Windows Forms. Для этого к производным элементам управления добавляют новый код либо переопределяют члены, унаследованные от базового класса. Предположим, что разработчику требуется создать текстовое поле, куда пользователь сможет вводить только числа. Для этого необходимо переопределить метод *OnKeyPress* следующим образом:

Visual Basic .NET

```
Public Class NumberBox
   Inherits System.Windows.Forms.TextBox
   Protected Overrides Sub OnKeyPress(ByVal e As KeyPressEventArgs)
      If Char.IsNumber(e.KeyChar) = False Then
         e.Handled = True
      End If
   End Sub
End Class
```

Visual C#

```
public class NumberBox : System.Windows.Forms.TextBox
{
   protected override void OnKeyPress(KeyPressEventArgs e)
   {
      if (char.IsNumber(e.KeyChar) == false)
         e.Handled = true;
   }
}
```

Модификация внешнего вида существующих элементов управления

Другая причина, по которой создают производные элементы управления, — их нестандартный внешний вид. Чтобы изменить стиль оформления существующего элемента управления, переопределите метод *OnPaint,* поместив в него собственный код. Например, чтобы изменить форму элемента управления, установите соответствующим образом его свойство *Region* в методе *OnPaint.* Класс *Region,* подобно *GraphicsPath,* описывает область экрана заданной формы, его получают из класса *GraphicsPath.* Так, можно сгенерировать класс *GraphicsPath,* представляющий форму вашего элемента управления, создать на его основе объект *Region* и поместить тот в соответствующее свойство элемента управления. Этот прием иллюстрируется на примере создания кнопки в форме текста:

Visual Basic .NET

```
Public Class WowButton
   Inherits System.Windows.Forms.Button
   Protected Overrides Sub OnPaint(ByVal pe As PaintEventArgs)
      Dim myPath As New Drawing2D.GraphicsPath()
      ' Создать объект GraphicsPath и установить свойства FontFamily
      ' и FontStyle элемента управления так, чтобы GraphicsPath представлял
      ' контуры строки символов размером 72 пункта.
      myPath.AddString("Wow!", Font.FontFamily, Font.Style, 72, New _
         PointF(0, 0), StringFormat.GenericDefault)
      ' Создать на основе GraphicsPath новый объект Region.
      Dim myRegion As New Region(myPath)
      ' Поместить новый объект Region в одноименное свойство
      ' элементауправления,
      Me.Region = myRegion
   End Sub
End Class
```

Visual C#

```
public class WowButton : System.Windows.Forms.Button
{
   protected override void OnPaint(PaintEventArgs pe)
   {
      System.Drawing.Drawing2D.GraphicsPath myPath = new
```

```
        System.Drawing.Drawing2D.GraphicsPath();
    // Следующая команда устанавливает свойства FontFamily и FontStyle
    // элемента управления так, чтобы объект GraphicsPath представлял
    // контуры строки символов размером 72 пункта. Учтите: в C# необходимо
    // явно преобразовать значение Font.Style в целое число.
    myPath.AddString("Wow!", Font.FontFamily, (int)Font.Style, 72,
        new PointF(0, 0), StringFormat.GenericDefault);
    // Создать на основе GraphicsPath новый объект Region.
    Region rcyRegion = new Region(myPath);
    // Поместить новый объект Region в одноименное свойство
    // элемента управления.
    this.Region = myRegion;
  }
}
```

Заметьте; *WowButton* — это не прямоугольная кнопка с прозрачным фоном, отображающая крупные буквы, она действительно имеет форму букв, составляющих слово «Wow!». Чтобы ее щелкнуть, необходимо попасть в область, ограниченную контуром букв. Сходным образом изменяют форму обычных элементов управления, чтобы получить элементы пользовательского интерфейса с нестандартным видом.

Примечание Прорисовка некоторых элементов управления, таких, как *TextBox*, выполняется не самим элементом управления, а формой, на которой он располагается. Такие элементы управления никогда не генерируют событие *Paint*, поэтому отвечающий за визуализацию код. помещенный в обработчик этого события, вызван не будет.

► **Создание производного элемента управления**

1. Объявите новый класс элемента управления на основе существующего класса элемента управления.
2. Реализуйте все необходимые нестандартные возможности и добавьте код, формирующий интерфейс производного элемента управления.

Создание составных элементов управления

Пользовательские, или составные, элементы управления объединяют нестандартные возможности с функциональностью стандартных элементов управления Windows Forms, позволяя быстро создавать новые элементов управления. Пользовательский элемент управления состоит из одного или нескольких стандартных элементов управления, объединенных в группу. Дизайнер UserControl предназначен для добавления к составным элементам управления дочерних элементов управления, также называемых вложенными. На рис. 7-1 показан дизайнер UserControl, в котором открыт составной элемент управления с парой вложенных кнопок и текстовым полем.

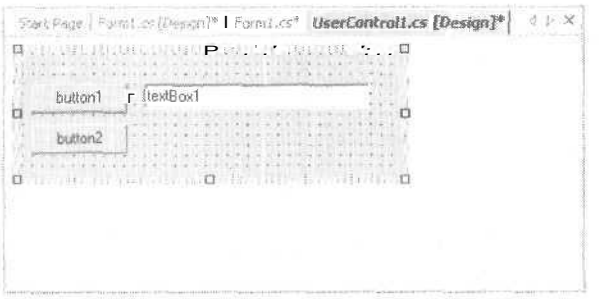


Рис. 7-1. Дизайнер UserControl

Разработчик может написать собственный код для обработки событий, генерируемых вложенными элементами управления. Предположим, что вам требуется создать пользовательский элемент управления, который состоит из двух текстовых полей и надписи. При изменении значений любого из текстовых полей этот элемент управления должен автоматически складывать их и отображать сумму в надписи. Подобную функциональность удается реализовать, переопределив методы *OnKeyPress* вложенных текстовых полей следующим образом:

Visual Basic .NET

```
' Для события KeyPress элемента управления TextBox2
' нужно создать аналогичный обработчик,
Private Sub TextBox1_KeyPress(ByVal Sender As Object, ByVal e As _
   KeyPressEventArgs) Handles TextBox1.KeyPress
   ' Проверить, является ли нажатая клавиша цифровой,
   If Char.IsNumber(e.KeyChar) = False Then
      e.Handled - True
   End If
   Label1.Text = (Integer.Parse(TextBox1.Text) + _
      Integer.Parse(TextBox2.Text)).ToString()
End Sub
```

Visual C#

```
// Для события KeyPress элемента управления TextBox2
// нужно создать аналогичный обработчик.
protected override void OnKeyPress(object sender, KeyPressEventArgs e)
   {
   // Проверить, является ли нажатая клавиша цифровой.
   if (char.IsNumber(e.KeyChar) == false)
      e.Handled - true;
   LabeU.Text = (int.Parse(TextBox1.Text) +
      int.Parse(TextBox2.Text)).ToString();
}
```

Вложенные компоненты составных элементов управления считаются закрытыми. Поэтому ни один разработчик, которому случится работать с вашим элементом управления, не сможет изменить такие свойства его вложенных элементов управления, как цвет, форма и размер. Чтобы разрешить другим разработчикам изменять

свойства вложенных элементов управления, сделайте их доступными через свойства объекта составного элемента управления. Рассмотрим составной элемент управления с вложенной кнопкой *(Button 1)*. Чтобы предоставить доступ к свойству *BackColor* объекта *Button1*, создайте, например, для него свойство-оболочку у объекта элемента управления, как показано ниже:

Visual Basic .NET

```
' Это свойство, объявленное в классе составного элемента управления.
Public Property ButtonColor() As Color
   Get
      Return Button1.BackColor
   End Get
   Set(ByVal Value As Color)
      Button1.BackColor = Value
   End Set
End Property
```

Visual C#

```
public color ButtonColor
{
   get
   {
      return Button1.BackColor;
   }
   set
   {
      Button1.BackColor = value;
   }
}
```

Чтобы предоставить доступ к вложенному элементу управления, измените его свойство *Modifiers*.Это возможно только через окно Properties в период разработки, поскольку в период выполнения оно не существует, Свойство *Modifiers* позволяет задать любой уровень доступа, который применяется к соответствующему вложенному элементу управления.

▶ **Как создать составной элемент управления**

1. Объявите класс, производный от *UserControl*.
2. Вызовите дизайнер UserControl и добавьте с его помощью вложенные элементы управления Windows Forms, после чего соответствующим образом настройте их.
3. При необходимости сделайте доступными свойства вложенных элементов управления.
4. Реализуйте нестандартные возможности, необходимые составному элементу управления.

Создание формируемых элементов управления

Формируемые элементы управления предоставляют максимум возможностей по настройке и управлению своей функциональностью, но их разработка занимает боль-

ше всего времени. Поскольку базовый класс *Control* не предоставляет своим потомкам даже базового графического интерфейса, разработчику придется самостоятельно писать весь код, формирующий облик такого элемента управления. Если интерфейс сложный, его создание занимает львиную долю цикла разработки элемента управления.

Процесс визуализации элемента управления в клиентской области формы называется *прорисовкой*. Получив команду на прорисовку, элемент управления генерирует событие *Paint*, при этом исполняются все обработчики события *Paint*. У класса *Control* обработчиком по умолчанию для события *Paint* является метод *OnPaint*.

Метод *OnPaint* принимает единственный аргумент — экземпляр класса *PaintEventArgs*, содержащий информацию клиентской области элемента управления. Стоит обратить внимание на два члена этого класса — *Graphics* и *ClipRectangle*.

Graphics — это объект типа *Graphics*, представляющий клиентскую область элемента управления. Ссылка на него необходима для формирования изображения элемента управления. *ClipRectangle*— прямоугольник, представляющий доступную клиентскую область элемента управления. При первом отображении элемента управления *ClipRectangle* ограничивает занятую им область. Если она перекрывается с другими элементам управления, данный элемент может оказаться частично или полностью скрытым. При повторной его прорисовке *ClipRectangle* представляет только видимую область, которая должна быть перерисована. Именно поэтому не рекомендуется определять размеры элемента управления по объекту *ClipRectangle*, для этой цели следует применять свойство *Size*.

По умолчанию координаты отсчитываются от верхнего левого угла элемента управления, в котором располагается условная точка начала координат (0,0), они исчисляются в пикселах. Следующий пример демонстрирует простой метод *OnPaint*, выполняющий прорисовку нестандартного элемента управления в виде красного эллипса:

Visual Basic .NET

```
' Этот пример предполагает наличие оператора Imports System.Drawing
Protected Overrides Sub OnPaint(ByVal pe As PaintEventArgs)
   Dim aBrush As New SolidBrush(Color.Red)
   Dim clientRectangle As New Rectangle(New Point(0,0), Me.Size)
   pe.Graphics.FillEllipse(aBrush, clientRectangle)
End Sub
```

Visual C#

```
// Этот пример предполагает наличие оператора Imports System.Drawing
protected override void OnPaint(PaintEventArgs e)
{
   Brush aBrush = new SolidBrush(Color.Red);
   Rectangle clientRectangle = new
      Rectangle(new Point(0,0), this.Size);
   e.Graphics.FillEllipse(aBrush, clientRectangle);
}
```

При изменении размеров элемента управления автоматически изменяется размер *ClipRectangle*, но повторная прорисовка этого элемента управления требуется не всегда. Чтобы при изменении размеров элемента управления он каждый раз про-

рисовывался заново, вызовите в его конструкторе методом *Control.SetStyle* и установите флаг *ResizeRedraw в true,* как показано в следующем примере:

Visual Basic .NET

```
SetStyle(ControlStyles.ResizeRedraw, True)
```

Visual C#

```
SetStyle(ControlStyles.ResizeRedraw, true);
```

Чтобы в любое время перерисовать элемент управления вручную, достаточно вызвать метод *Refresh:*

Visual Basic .NET

```
Refresh()
```

Visual C#

```
Refresh();
```

▶ **Как создать формируемый элемент управления**

1. Объявите класс, производный от класса *Control.*
2. Добавьте в метод *OnPaint* код, визуализирующий элемент управления.
3. Реализуйте все нестандартные возможности, необходимые вашему элементу управления.

Резюме

- Все элементы управления являются прямыми или косвенными потомками класса *Control,* предоставляющего им поддержку обработки событий клавиатуры и мыши, а также набор свойств, общих для всех элементов управления.
- Существует три типа нестандартных элементов управления: производные, составные и формируемые.
- Производные элементы управления создают путем наследования от класса стандартного элемента управления Windows Forms, при этом новый элемент управления сохраняет облик и функциональность базового класса.
- Составные элементы управления инкапсулируют один или несколько стандартных элементов управления Windows Forms и объявляются на основе класса *UserControl.* Их вложенные элементы управления по умолчанию закрыты и не доступны в период выполнения, если не открыть к ним доступ, изменив значение свойства *Modifiers.*
- Разработка формируемых элементов управления требует больше всего времени. Эти элементы управления являются потомками класса *Control,* который не предоставляет никакого интерфейса. Поэтому разработчик отвечает не только за реализацию функциональности элемента управления, ни и за написание кода, выполняющего его прорисовку.

Занятие 3. Решение типичных задач, связанных с разработкой элементов управления

При разработке любых типов элементов управления возникают различные задачи, одни чаще, другие реже. На этом занятии вы научитесь решать типичные задачи, сопутствующие разработке элементов управления.

Изучив материал этого занятия, вы сможете:

- ✓ добавить свой элемент управления на инструментальную панель Toolbox;
- ✓ назначить своему элементу управления значок для отображения на панели Toolbox;
- ✓ отладить созданный элемент управления;
- ✓ определить политику лицензирования своих элементов управления;
- ✓ просматривать элемент управления с помощью Internet Explorer.

Продолжительность занятия - около 20 минут.

Добавление элементов управления на панель Toolbox

Удобно, когда собственные элементы управления всегда под рукой. Чтобы сделать их доступными при разработке любых проектов и упростить работу с ними, добавьте их на инструментальную панель Toolbox, откуда их в любое время можно перетащить на форму.

Эту задачу позволяет решить команда Customize Toolbox, доступная через контекстное меню панели Toolbox. Щелкните правой кнопкой панель Toolbox и выберите команду Customize Toolbox, чтобы вызвать диалоговое окно Customize Toolbox. В нем найдите и выберите файл с элементом управления, который нужно добавить на панель Toolbox.

► **Добавление элемента управления на инструментальную панель Toolbox**

1. Щелкните правой кнопкой мыши панель Toolbox и выберите команду Customize Toolbox — откроется одноименное диалоговое окно.
2. Перейдите на вкладку .NET Framework Components и щелкните кнопку Browse, чтобы вызвать диалоговое окно File.
3. Откройте папку с DLL- или EXE-файлом, в котором находится требуемый элемент управления. Выделите нужный файл и щелкните кнопку Open.
4. Убедитесь, что выбранный элемент управления добавлен к списку диалогового окна Customize Toolbox и щелкните OK — элемент управления появится на панели Toolbox.

Назначение значка элементу управления

Visual Studio .NET предоставляет значок для отображения нестандартных элементов управления, добавленных на панель Toolbox, но можно назначить для этой цели и собственное растровое изображение. Далее подробно рассказано, как это сделать.

Значок для отображения на панели Toolbox назначают с помощью класса *ToolboxBitmapAttribute* — специализированного атрибута, предоставляющего метаданные элемента управления. Он позволяет назначить элементу управления растровое изоб-

ражение размером 16x16 пикселов либо задать значок по значению *Type (type)*. В последнем случае элементу управления присваивается значок, принятый для объекта данного типа.

Атрибут *ToolboxBitmapAttribute* входит в объявление класса элемента управления, В Visual Basic .NET его заключают в угловые скобки (<>) и помещают в одной строке с объявлением класса, перед собственно объявлением. В Visual C# этот атрибут заключают в квадратные скобки и также располагают в строке объявления класса элемента управления, перед самим объявлением.

► **Как назначить элементу управления значок, указав файл с его изображением**

Укажите файл с изображением значка при помощи атрибута *ToolboxBitmapAttribute*:

Visual Basic .NET

```
<ToolboxBitmap("C:\Pasta.bmp")> Public Class PastaMaker
    ' Реализация опущена.
End Class
```

Visual C#

```
[ToolboxBitmap(@"C:\Pasta.bmp")]
public class PastaMaker : Control
{
   // Реализация опущена.
}
```

► **Как назначить элементу управления значок на основе его типа**

Добавьте к объявлению класса элемента управления атрибут *ToolboxBitmapAttribute* со ссылкой на тип данного элемента управления:

Visual Basic .NET

```
<ToolboxBitmap(GetType(Button))> Public Class myButton
    ' Реализация опущена.
End Class
```

Visual C#

```
[ToolboxBitmap(typeof(Button))]
public class myButton : Button
{
   // Реализация опущена.
}
```

Отладка элементов управления

В цикле разработки любого компонента рано или поздно наступает этап отладки, и элементы управления тут не исключение. Поскольку они не являются проектами, для отладки их необходимо поместить в специальный проект Windows Forms,

Перед тестированием элемента управления его необходимо скомпилировать. После компоновки проекта, в котором находится ваш элемент управления, его можно помешать на формы и отлаживать, как обычно.

Если ваш элемент управления является частью проекта исполняемого компонента, например приложения Windows Forms, для ето тестирования следует добавить к проекту новую форму, а если элемент управления входит в проект библиотеки классов или элементов управления, придется добавить к решению дополнительный проект.

▶ **Отладка элемента управления в проекте Windows Forms**

1. Выполните компоновку решения, выбрав в меню Build команду Build Solution.
2. При необходимости добавьте в проект новую форму и сделайте ее начальной формой проекта.
3. Добавьте элемент управления на форму через графический интерфейс либо программно. В первом случае сначала добавляют элемент управления на панель Toolbox, чтобы с ним было проще работать в период разработки (см. выше),
4. Запустите приложение, нажав F5, — теперь можно приступать к отладке элемента управления: установите точки прерывания, применяйте пошаговое исполнение и другие отладочные средства, позволяющие находить ошибки в коде.
5. Исправив ошибки, перекомпилируйте элемент управления, чтобы внесенные в его код изменения вступили в силу. Для этого выберите в меню Build команду Rebuild Solution.

▶ **Отладка элемента управления в проекте библиотеки классов или элементов управления**

1. Выполните компоновку решения, выбрав в меню Build команду Build Solution.
2. В меню File выберите команду Add Project\New Project, чтобы добавить к решению новый проект, — откроется диалоговое окно Add New Project.
3. В диалоговом окне Add New Project выберите Windows Application, дайте проекту имя и щелкните OK.
4. В окне Solution Explorer щелкните правой кнопкой мыши узел нового проекта References и выберите из контекстного меню команду Add Reference — откроется диалоговое окно Add Reference.
5. Перейдите на вкладку Projects. Если проект, содержащий элемент управления, указан на этой вкладке, выделите его и щелкните (Ж, в противном случае щелкните кнопку Browse и найдите папку с DLL-файлом проекта, в котором находится отлаживаемый элемент управления. Выделите его и щелкните OK.
6. Добавьте элемент управления на форму через графический интерфейс либо программно. В первом случае иногда сначала приходится добавить элемент управления на панель Toolbox, упростить работу с ним в период разработки (см, выше).
7. Запустите приложение, нажав F5 — теперь можно приступать к отладке элемента управления: установите точки прерывания, примените пошаговое исполнение и другие отладочные средства, позволяющие находить и устранять ошибки в коде.

Управление лицензированием элементов управления

В .NET Framework имеется встроенная поддержка управления лицензированием элементов управления. Лицензирование элементов управления позволяет их создателям защитить свою интеллектуальную собственность: предоставить доступ к своим элементам управления только авторизированным разработчикам. Модель лицен-

зирования, принятая по умолчанию, требует, чтобы все лицензируемые элементы управления были помечены атрибутом *License ProviderAttribute*, указывающим объект *License Provider* — он необходим для проверки подлинности лицензии. *License Provider* — это абстрактный класс, предоставляющий стандартный интерфейс проверки подлинности. В конструкторе элемента управления следует вызвать метод *LicenseManager.Validate*, чтобы получить ссылку на действительную лицензию. Если лицензия на элемент управления отсутствует, его загрузка закончится неудачей.

Метод *License Manager. Validate* вызывает метод *GetLicense* заданного объекта *License Provider,* который извлекает лицензию и проверяет ее при помощи метода *IsKey Valid*объекта *License Provider.* Если *IsKey Valid* возвращает *true*, то лицензия считается действительной, в противном случае этот элемент управления не загружается. Применяемая схема проверки зависит от реализации *License Provider.* .NET- реализация класса *License Provider* называется *Lic File License Provider.* Если этот класс задан как *License Provider,* он применяется для проверки лицензии. Метод *GetLicense* этого класса ищет в каталоге с DLL-файлом элемента управления текстовый файл с именем *FullName.LIC*, где *FullName* — полное имя элемента управления. Затем *IsKey Valid* проверяет наличие в первой строке текстового файла подстроки «myClassName is a licensed component», где myClassName — полное имя компонента. Разработчик волен заменить этот алгоритм проверки собственным, переопределив соответствующие методы. В каждом лицензируемом элементе управления должен быть метод *Dispose,* который следует вызывать для освобождения ресурсов, связанных с проверкойлицензии.

Вот пример лицензируемого элемента управления *Widget,* использующего проверку с помощью класса *Lic File License Provider.*

Visual Basic .NET

```
' Этот пример предполагает наличие оператора Imports System.ComponentModel.
' Это атрибут LicenseProvider, указывающий тип применяемого LicenseProvider,
<LicenseProvider(GetType(LicFileLicenseProvider))> Public Class _
   Widget
   Inherits Control
   Private myLicense As License
   Public Sub New()
      ' Проверить лицензию и получить ссылку на нее.
      myLicense = LicenseManager.Validate(GetType(Widget), Me)
      ' Остальной код конструктора опущен.
   End Sub
   ' Это метод Dispose, освобождающий ресурсы, связанные с проверкой лицензии.
   Protected Overloads Overrides Sub _
      Dispose(ByVal Disposing As Boolean)
      If Not (myLicense Is Nothing) then
         myLicense.Dispose()
         myLicense = Nothing
      End If
   End Sub
End Class
```

Visual C#

```
// Этот пример предполагает наличие оператора Imports System.ComponentModel.
// Это атрибут LicenseProvider, указывающий тип применяемого LicenseProvider.
 [LicenseProvider(typeof(LicFileLicenseProvider))]
public class Widget : System.Windows.Forms.Control
{
   private License myLicense;
   public Widget()
   {
      // Проверить лицензию и получить ссылку на нее.
      myLicense = LicenseManager.Validate(typeof(Widget), this);
      // Остальной код конструктора опущен.
   }
   // Это метод Dispose, освобождающий ресурсы, связанные с проверкой лицензии.
   protected override void Dispose(bool Disposing)
   {
      if (myLicense != null)
      {
         myLicense.Dispose();
         myLicense = null;
      }
   }
}
```

▶ **Реализация управления лицензированием элемента управления**

1. Укажите класс *LicenseProvider,* добавив к объявлению класса атрибут *LicenseProviderAttribute.*
2. Создайте файл License, требуемый выбранным классом *LicenseProvider,*
3. В конструкторе элемента управления вызовите метод *LicenseManager.Validate* для проверки лицензии.
4. Освободите ресурсы, занятые проверкой лицензии, вызвав метод *Dispose* элемента управления.

Загрузка элемента управления в Internet Explorer

Любой элемент управления Windows Forms можно загрузить в Internet Explorer. Это позволяет обогащать HTML-страницы широкими возможностями таких элементов управления. Из этого раздела вы узнаете, как это проделать.

Чтобы загрузить элемент управления в Internet Explorer, он должен располагаться в кэше глобальных сборок (Global Assembly Cache, GAC) либо в одном виртуальном каталоге с HTML-страницей, в которой он объявлен.

Примечание Об установке классов в GAC рассказывается в главе 9.

Для объявления элементов управления Windows Forms в HTML-страницах служит тэг <OBJECT>, который приказывает вставить в страницу компилированный объект. Тэг <OBJECT> определяет тип загружаемого объекта по его свойству *classid*, поэтому можно задать загружаемый элемент управления Windows Forms через это свойство.

Свойство *classid* элемента управления Windows Forms состоит из двух частей: пути к файлу элемента управления и его полного имени, разделенных знаками «#». Ниже показан пример тэга <OBJECT>, объявляющего элемент управления с полным именем *ControlLibrary1.myControl*, извлекаемый из файла ControlLibrary1.dll, который расположен в одном виртуальном каталоге с HTML-страницей:

```
<OBJECT id="myControl"
classid="http:ControlLibrary1.dll#ControlLibrary1.myControl
VIEWASTEXT>
</OBJECT>
```

▶ **Загрузка элемента управления в Internet Explorer**

1. Запишите DLL-файл элемента управления в виртуальный каталог HTML-страницы, в которой он объявлен, или установите его в GAC.
2. Добавьте к HTML-странице тэг <OBJECT> с соответствующим значением *classid*. Свойство *classid* элемента управления включает путь к его DLL-файлу и полное имя, разделенные знаком «#».

Резюме

- Чтобы добавить элемент управления на панель Toolbox, щелкните ее правой кнопкой мыши и выберите из контекстного меню команду Customize Toolbox — элемент управления появится на панели инструментария Toolbox, ему будет назначен значок по умолчанию. Чтобы назначить элементу управления другой значок, воспользуйтесь атрибутом *ToolboxBitmapAttribute*.
- Для отладки элемента управления его необходимо сначала скомпилировать и разместить на форме. Если отлаживаемый элемент управления входит в исполняемый проект, для тестирования к проекту необходимо добавить новую форму, в противном случае следует добавить тестовый проект. После этого для отладки элемента управления можно использовать обычные отладочные средства Visual Studio. Исправив ошибки, необходимо перекомпилировать элемент управления, чтобы изменения вступили в силу.
- .NET Framework поддерживает модель лицензирования элементов управления. Чтобы задействовать ее, укажите класс *License Provider*, добавив к классу элемента управления атрибут *Lic File License Provider*, затем вызовите метод *License Manager.Validate*, чтобы получить ссылку на лицензию. Закончив, освободите ресурсы, занятые проверкой лицензии.
- Чтобы загрузить элемент управления в Internet Explorer, объявите его с помощью тэга <OBJECT>. Тэг <OBJECT> проверяет указанное в нем значение свойства *classid*, чтобы определить загружаемый элемент управления.

Практикум 7. Создание нестандартного элемента управления

Ваша задача — создать нестандартный элемент управления *PrettyClock*, отображающий на форме красиво оформленные часы. Вы напишете реализацию и код, визуализирующий этот элемент управления, а также добавите в его конструктор код, инициализирующий свойства этого элемента управления. В завершение вы протестируете созданный элемент управления. Решение задач этого практикума находится в папке \Labs\Ch07\ Solution на прилагаемом к книге компакт-диске.

Прежде всего

Для выполнения этого практикума предварительная подготовка не требуется.

Продолжительность практикума - около 30 минут.

Упражнение 1. Создание элемента управления

Сейчас вы создадите нестандартный элемент управления *PrettyClock*. Он относится к формируемым элементам управления, поскольку вы объявите его на основе класса *Control*, самостоятельно напишете код, отвечающий за визуализацию этого элемента управления, и откроете доступ к свойствам, управляющим его обликом. В завершение вы добавите таймер, чтобы ваши часы могли служить по прямому назначению.

▶ Создание проекта

1. В меню File выберите команду New\Project — откроется диалоговое окно New Project.
2. В окне New Project выберите шаблон Windows Application, назовите новый проект **ControlTest.vb** (или **ControlTest.cs**) **и** щелкните OK.
3. В меню Project выберите команду Add New Item — откроется одноименное диалоговое окно.
4. В диалоговом окне Add New Item выберите Custom Control, назовите новый элемент управления соответственно **PrettyClock.vb или** (**PrettyClock.cs**) и щелкните Open — к проекту будет добавлен новый нестандартный элемент управления.
5. В окне Solution Explorer щелкните правой кнопкой элемент управления PrettyClock и выберите из контекстного меню команду View Code — откроется окно Code. Обратите внимание, что метод *OnPaint* переопределен и содержит вместо реализации заглушку и комментарий, указывающий, что сюда нужно добавить код, выполняющий визуализацию.

Теперь пора писать код. Наш элемент управления будет отображать текущее время в прямоугольной области экрана. Для его прорисовки применяется метод *LinearGradientBrush*, поддерживающий интересные графические эффекты. Вы также реализуете у своего элемента управления свойства, позволяющие разработчикам управлять его обликом *E* период выполнения.

1. В окне редактора кода поместите в начало кода следующие операторы, импортирующие пространства имен *System.Drawing* и *System.Drawing.Drawing2D*:

Visual Basic .NET

```
Imports System.Drawing
Imports System.Drawing.Drawing2D
```

Visual C#

```
// В C# опеэатор 'using System.Drawing;' добавлен по умолчанию
using System.Drawing.Drawing2D;
```

2. Добавьте свойства, хранящие описание цвета. Оно, вместе со свойством *ForeColor*, унаследованным от базового класса, предназначено для создания объекта *LinearGradientBrush*, визуализирующего нестандартный элемент управления. Вот примерный код этого свойства:

Visual Basic .NET

```
' Объявить закрытую переменную для хранения значения Color.
Private mForeColorTwo As Color
' Это реализация свойства.
Public Property ForeColorTwo() As Color
    Get
        Return mForeColorTwo
    End Get
    Set(ByVal Value As Color)
        mForeColorTwo - Value
    End Set
End Property
```

Visual C#

```
// Обьявить закрытую переменную для хранения значения Color,
private Color mForeColorTwo;
// Это реализация свойства.
public Color ForeColorTwo
{
    get
    {
        return mForeColorTwo;
    }
    set
    {
        mForeColorTwo = value;
    }
}
```

3. Добавьте свойство *Angle* типа *Single (float)* для хранения значения угла, необходимого методу *LinearGradientBrush*. Поскольку его значение выражается в градусах, потребуется код, проверяющий, что введенное значение не меньше 0 и не больше 360. Вот код этого свойства:

Visual Basic .NET

```
Private mAngle As Single
Public Property Angle() As Single
   Get
      Return mAngle
   End Get
   Set(8yVal Value As Single)
      ' Такая схема устанавливает минимальное значение при попытке
      '  ввести слишком большее значение и наоборот. Это обеспечивает
      ' непрерывное приращение значения свойства.
      If Value > 360 then
         mAngle = 0
      ElseIf Value < 0 then
         mAngle = 360
      Else
         mAngle = Value
      End If
   End Set
End Property
```

Visual C#

```
private float mAngle;
public float Angle
{
   get
   {
      return mAngle;
   }
   set
   {
      // Такая схема устанавливает минимальное значение при попытке
      // ввести слишком большее значение и наоборот. Это обеспечивает
      // непрерывное приращение значения свойства.
      if (value > 360)
         mAngle = 0;
      else if (value < 0)
         mAngle = 360;
      else
         mAngle = value;
   }
}
```

4. Добавьте в метод *OnPaint* код, создающий объект *LinearGradientBrush* и отображающий текущее время с помощью объекта *Font*. Для создания объекта *LinearGradientBrush* вам потребуются свойства *Size*, *ForeColor* и *ForeColorTwo*, которые можно использовать так:

Visual Basic .NET

```
' Создать представляющий элемент управления прямоугольник, заданный
' точкой верхнего левого угла и свойством Size.
Dim ControlRectangle As New Rectangle(New Point(0, 0), Me.Size)
' Создать новый объект LinearGradientBrush, определяющий переход
' между цветами ForeColor и ForeColorTwo с учетом заданного угла,
Dim myBrush As New LinearGradientBrush(ControlRectangle, _
   ForeColor, ForeColorTwo, Angle)
' Создать объект GraphicsPath, хранящий отображаемый текст,
Dim myPath As New GraphicsPath()
' Записать в объект GraphicsPath строку, представляющую текущее время.
myPath.AddString(Now.ToLongTimeString(), Font.FontFamily, _
   Font.Style, Font.Size, New Point(0, 0), _
   StringFormat.GenericDefault)
' Показать Graphics.
pe.Graphics.FillPath(myBrush, myPath)
```

Visual C#

```
// Создать представляющий элемент управления прямоугольник, заданный
// точкой верхнего левого угла и свойством Size.
Rectangle ControlRectangle = new
   Rectangle(new Point(0, 0), this.Size);
// Создать новый объект LinearGradientBrush, определяющий переход
// между цветами ForeColor и ForeColorTwo с учетом заданного угла.
LinearGradientBrush myBrush = new
   LinearGradientBrush(ControlRectangle,
   ForeColor, ForeColorTwo, Angle);
// Создать объект GraphicsPath, хранящий отображаемый текст.
GraphicsPath myPath = new GraphicsPathO;
// Записать в объект GraphicsPath строку, представляющую текущее время.
myPath.AddString(System.DateTime.Now.ToLongTimeString(),
   Font.FontFamily,
   (int)Font.Style, Font.Size, new Point(0, 0),
   StringFormat.GenericDefault);
// Показать Graphics.
pe.Graphics.FillPath(myBrush, myPath);
```

5. Теперь поместите код для создания нового пера из объекта *LinearGradientBrush*. Прорисуйте этим пером прямоугольник, представляющий область элемента управления. Для этого потребуется следующий код:

Visual Basic .NET

```
' Создать из объекта myBrush перо толщиной 8 пикселов
Dim myPen As New Pen(myBrush, 8)
' и нарисовать им прямоугольник, совпадающий с областью элемента управления.
pe.Graphics.DrawRectangle(myPen, ControlRectangle)
```

Visual C#

```
// Создать из объекта myBrush перо толщиной 8 пикселов
Pen myPen = new Pen(myBrush, 8);
// и нарисовать им прямоугольник, совпадающий с областью элемента управления.
pe.Graphics.DrawRectangle(myPen, ControlRectangle);
```

6. Выберите в меню File команду Save All, чтобы сохранить проект.

Сейчас вы добавите в конструктор код, инициализирующий свойства элемента управления. Кроме того, вы добавите компонент *Timer* и код для ежесекундной перерисовки часов и отображения интересного графического эффекта.

▶ **Настройка компонента *Timer***

1. В окне Solution Explorer щелкните правой кнопкой компонент *PrettyClock* и выберите из контекстного меню команду View Designer — откроется окно дизайнера.
2. Перетащите компонент *Timer* с вкладки Windows Forms панели Toolbox на поверхность формы.
3. В окне Properties установите свойства *Interval* в *1000,* а свойство *Enabled* ~ *в true.*
4. Дважды щелкните компонент *Timer* в окне дизайнера, чтобы открыть обработчик событий *Timer.Tick,* и добавьте к нему следующий код:

Visual Basic .NET

```
' Этот кода изменяет вид графического эффекта при изменении значения Tick.
Angle += 10
' Перерисовать элемент управления.
Me.Refresh()
```

Visual C#

```
// Этот кода изменяет вид графического эффекта при изменении значения Tick.
Angle += 10;
// Перерисовать элемент управления.
this.Refresh();
```

5. В Visual C# необходимо добавить к конструктору элемента управления вызов метода *InitializeComponent*, инициализирующего свойства этого элемента управления:

Visual C#

```
public PrettyClock()
    {
        InitializeComponent();
    }
```

6. Сохраните проект, выбрав команду Save All в меню File.

Упражнение 2. Тестирование элемента управления

Создав элемент управления, необходимо удостовериться, что он работает правильно. Сейчас вы добавите элемент управления на панель Toolbox и поместите его на форму. Затем вы настроите вид элемента управления на панели Toolbox и проверите его работу в период выполнения.

▶ Как добавить элемент управления на панель Toolbox

1. Выполните компоновку решения, выбрав в меню Build команду Build Solution.
2. Щелкните правой кнопкой панель Toolbox и выберите из контекстного меню команду Customize Toolbox — откроется одноименное диалоговое окно Customize Toolbox.
3. Перейдите на вкладку .NET Framework Components и щелкните кнопку Browse, чтобы открыть диалоговое окно Open File.
4. Откройте папку, в которой находится проект. В папке \bin (Visual Basic .NET) или \bin\Debug (Visual C#) выберите файл ControlTest.exe (компилированный файл элемента управления). Чтобы продолжить, щелкните OK.
5. Убедитесь, что элемент управления *PrettyClock* указан в списке диалогового окна Customize Toolbox и помечен флажком. Щелкните OK.

▶ Как протестировать элемент управления

1. В окне Solution Explorer щелкните правой кнопкой форму *Form1* и выберите из контекстного меню команду View Designer — эта форма откроется в окне дизайнера.
2. Перетащите элемент управления *PrettyClock* с панели Toolbox на форму — к форме добавится новый экземпляр этого элемента управления.
3. Установите для свойств нового экземпляра *PrettyClock*, перечисленных в таблице 7-8, необходимые значения.

Таблица 7-8. Свойства элемента управления *PrettyClock*

Свойство	Значение
ForeColor	Red
ForeColorTwo	Yellow
Font.Size	40
Size	225, 50

Обратите внимание, что при модификации значений свойств изменения вступают в силу немедленно и результаты сразу же отображаются на форме.

4. Убедитесь, что форма обновляется ежесекундно, изменяя вид цветового перехода. Для этого нажмите F5, чтобы запустить приложение. Заметьте, что в период выполнения поведение элемента управления не изменилось,

Закрепление материала

Приведенные ниже вопросы помогут вам лучше усвоить основные темы данной главы. Если вы не сумеете ответить на вопрос, повторите материал соответствующего занятия. Ответы для самопроверки — в приложении «Вопросы и ответы» в конце книги.

1. Кратко опишите каждый из трех видов нестандартных элементов управления и укажите их отличия.

2. Расскажите о роли объектов *Graphics, Brush, Pen* и *GraphicsPath* в визуализации графических элементов.

3. Как нарисовать текст в клиентской области формы или другого объекта?

4. Какова роль объекта *LicenseProvider* в лицензировании элементов управления?

5. Как создать форму или элемент управления, форма которого будет отличаться от прямоугольной?

ГЛАВА 8

Сложные случаи программирования с применением .NET Framework

Занятие 1. Поддержка печати 344

Занятие 2. Получение доступа к компонентам и их вызов 357

Занятие 3. Реализация дополнительных возможностей доступа 365

Занятие 4. Создание справочной системы 370

Занятие 5. Глобализация и локализация 373

В этой главе

В этой главе рассматриваются сложные случаи программирования с применением Microsoft .NET Framework. Вы узнаете, как дополнить приложение поддержкой печати, получить доступ к внешним компонентам, сделать приложение удобным для пользователей и создать справочную систему, а также как реализовать поддержку различных языков и региональных стандартов.

Прежде всего

Для изучения этой главы необходимо изучить материал глав 1–7.

Занятие 1. Поддержка печати

В любой организации, независимо от рода ее деятельности, часто требуется печатать данные и документы. В .NET Framework предусмотрен компонент *PrintDocument*, обеспечивающий вывод на печать и предоставляющий ряд классов для поддержки печати и установки связанных с ней параметров. На этом занятии вы научитесь создавать и конфигурировать объект *Print Document, a* также печатать с его помощью отдельные строки, текстовые файлы и графику.

Изучив материал этого занятия, вы сможете:

- ✓ создать и сконфигурировать объект *PrintDocument*;
- ✓ вывести на печать текст;
- ✓ печатать графические элементы;
- ✓ применять элемент управления *PrintPreview* и диалоговое окно Print Preview;
- ✓ настраивать параметры страницы при помощи элемента управления *PageSetupDialog*;
- ✓ выбирать принтер с помощью элемента управления *PrintDialog*.

Продолжительность занятия – **около** 45 минут.

Без сомнения, возможность печати является одной из важнейших для деловых приложений. Несмотря на то, что псе больше людей хранят файлы и документы в электронном виде, вряд ли в ближайшем будущем они полностью откажутся от печатных материалов. Поэтому разработчикам необходимо уметь создавать поддержку печати в своих приложениях. .NET Framework предоставляет классы и компоненты, которые помогут вам просто и быстро дополнить свою программу этой функцией.

Компонент PrintDocument

В .NET Framework печатаемый документ представлен компонентом *PrintDocument*. Хотя у *PrintDocument* нет графического интерфейса, этот компонент находится на вкладке Windows Forms инструментальной панели Toolbox, откуда его можно перетащить на форму приложения, открытую в окне дизайнера.

Объект *PrintDocument* инкапсулирует всю информацию, необходимую для печати страниц. У него есть ряд свойств: *PrinterSettings* хранит сведения о возможностях и конфигурации доступных принтеров, *DefaulPageSettings* инкапсулирует параметры печати страниц, а *PrintController* контролирует обработку страниц при печати. Эти свойства позволяют весьма действенно управлять процессом печати.

Создание объекта PrintDocument

Объект *PrintDocument* можно создать в период разработки, с помощью графического интерфейса дизайнера, или программно, в период выполнения. В первом случае объект *PrintDocument* следует перетащить в окно дизайнера прямо с панели Toolbox — в области компонентов появится новый экземпляр этого объекта, автоматически сконфигурированный для работы с системным принтером по умолчанию.

Вот как программно создается экземпляр объекта *PrintDocument* :

Visual Basic .NET

```
Dim myPrintDocument As New PrintDocument()
```

Visual C#

```
PrintDocument myPrintDocument = new PrintDocumentO;
```

Полученный таким образом экземпляр объекта *PrintDocument* автоматически конфигурируется для работы с системным принтером по умолчанию.

Как работает поддержка печати

В модели печати, принятой в .NET Framework, печатаемые данные непосредственно передаются кодом приложения. Вызов метода *PrintDocument, Print* инициирует новое задание печати, попутно генерируя события *PrintPage*. Если клиентский обработчик для этого события отсутствует, печать не выполняется. Чтобы задать данные для печати, необходимо написать обработчик для события *PrintPage*.

Если задание печати состоит из нескольких страниц, событие *PrintPage* генерируется для каждой из них, поэтому обработчик этого события вызывается многократно. Обработчик события *PrintPage* контролирует исполнение задания печати, обеспечивая последовательный вывод многостраничного документа; без его участия печатались бы лишь несколько копий первой страницы документа.

Событие PrintPage

Событие *PrintPage* играет ключевую роль в процессе печати. Чтобы реально отправить данные на принтер, необходимо создать обработчик для события *PrintPage* и поместить в него код, прорисовывающий печатаемое содержимое. Все данные и объекты, необходимые для печати, передаются обработчику в объекте *PrintPage-EventArgs*, свойства которого описаны в таблице 8-1,

Таблица 8-1. Свойства объекта *PrintPageEventArgs*

Имя	Описание
Cancel	Указывает, отменено ли задание печати
Graphics	Содержит объект *Graphics,* формирующий печатаемую страницу
HasMorePages	Получает или устанавливает значение, указывающее, все ли страницы напечатаны
MarginBounds	Получает объект *Rectangle,* представляющий область страницы, ограниченную полями
PageBounds	Получает объект *Rectangle*, представляющий страницу целиком
PageSettings	Получает или устанавливает объект *PageSettings* для текущей страницы

Для преобразования содержимого документа в данные для принтера применяется объект *Graphics*, доступный через класс *Print Page EventArgs*. В этом случае печатаемая страница играет ту же роль, что и клиентская области формы, элемента управления и любой другой поверхности для рисования, представленной объектом *Graphics*. Печатаемую страницу формируют теми же методами, что применяются для прорисовки содержимого формы на экране. Вот пример очень простого метода, который печатает эллипс, вписанный в область, ограниченную полями страницы:

Visual Basic .NET

```
' Чтобы этот метод был вызван, он должен обрабатывать событие PrintPage.
Public Sub PrintEllipse(ByVal sender As System.Object, ByVal e As _
   System.Drawing.Printing.PrintPageEventArgs)
   e.Graphics.DrawEllipse(Pens.Black, e.MarginBounds)
End Sub
```

Visual C#

```
// Чтобы этот метод был вызван, он должен обрабатывать событие PrintPage.
public void PrintEllipse(object sender,
   System.Drawing.Printing.PrintPageEventArgs e)
{
   e.Graphics.DrawEllipse(Pens.Black, e.MarginBounds);
}
```

Свойства *MarginBounds* и *PageBounds* указывают области страницы, доступные для печати. Можно границами области печати сделать поля страницы, вычислив координаты печатаемых объектов на основе прямоугольника *MarginBounds*. Допустимо печатать и за пределами страничных полей — например, чтобы создать верхний и нижний колонтитулы. Для этого следует рассчитывать координаты объектов на основе прямоугольника *PageBounds*, Как и при выводе на дисплей, при печати координаты по умолчанию исчисляются в пикселах.

Свойство *HasMorePages* позволяет указать, является ли задание печати многостраничным. По умолчанию это свойство установлено в *false*. Если же потребуется напечатать многостраничный документ, следует установить его в *true*, а по завершении печати последней страницы — снова в *false*.

Примечание Обработчик события *PrintPage* должен следить за числом страниц задания печати, в противном случае результаты будут непредсказуемыми. Скажем, если по завершении вывода последней страницы не установить свойство *HasMorePages* в *false*, приложение будет генерировать события *PrintPage* снова и снова.

Кроме того, предусмотрена возможность отменить задание печати, не дожидаясь окончания вывода текущей страницы. Для этого следует установить свойство *Cancel* в *true*.

Чтобы создать обработчик события *PrintPage*, дважды щелкните экземпляр *PrintDocument* в окне дизайнера (так создается обработчик по умолчанию) либо объявите обработчик вручную (см. главу 3).

Вывод на печать

Чтобы создать задание печати, достаточно вызвать метод *Print Document, Print*. В следующем разделе вы узнаете, как написать код, который определяет тип содержимого, предназначенного для вывода на печать.

Печать графики

Печатать графические элементы не сложнее, чем рисовать их на экране. Для обеих целей применяют объект *Graphic*, доступный через объект *PrintPageEventArgs*. Печатают как простые фигуры, так и более сложные; в последнем случае используют

объект *GraphicsPath*. Вот как напечатать сложную фигуру с помощью объекта *GraphicsPath*:

Visual Basic .NET

```
' Этот метод должен обрабатывать событие PrintPage.
Public Sub PrintGraphics(ByVal Sender As System.Object, ByVal _
   e As System.Drawing.Printing.PrintPageEventArgs)
   Dim myPath As New System.Drawing.Drawing2D.GraphicsPath()
      myPath.AddPolygon(New Point() {New Point(1, 1), _
         New Point(12, 55), New Point(34, 8), New Point(52, 53), _
         New Point(99, 5)})
   myPath.AddRectangle(New Rectangle(33, 43, 20, 20))
   e.Graphics.DrawPath(Pens.Black, myPath)
End Sub
```

Visual C#

```
// Этот метод должен обрабатывать событие PrintPage.
public void PrintGraphics(object sender,
   System.Drawing.Printing.PrintPageEventArgs e)
{
   System.Drawing.Drawing2D.GraphicsPath myPath = new
      System.Drawing.Drawing2D.GraphicsPath();
      myPath.AddPolygon(new Point[] {new Point(1, 1). new
         Point(12, 55), new Point(34, 8), new Point(52, 53),
         new Point(99, 5)});
   myPath.AddRectangle(new Rectangle(33, 43, 20, 20));
   e.Graphics.DrawPath(Pens.Black, myPath);
}
```

Чтобы напечатать многостраничный графический документ, следует вручную разбить его на страницы и написать соответствующий код. Следующий метод разбивает при печати эллипс, не умещающийся на одной странице, на две страницы:

Visual Basic .NET

```
Dim FirstPagePrinted As Boolean = False
' Этот метод должен обработать событие PrintPage
Public Sub PrintBigEllipse(ByVal Sender As System.Object, ByVal _
   e As System.Drawing.Printing.PrintPageEventArgs)
   If FirstPagePrinted = False Then
      FirstPagePrinted = True
      e.HasMorePages = True
      e.Graphics.DrawEllipse(Pens.Black, New Rectangle(0, 0, _
         e.PageBounds.Width, e.PageBounds.Height * 2))
   Else
      e.HasMorePages = False
      FirstPagePrinted = False
      e.Graphics.DrawEllipse(Pens.Black, New Rectangle(0, _
         -(e.PageBounds.Height), e.PageBounds.Width, _
```

```
        e.PageBounds.Height * 2))
    End If
End Sub
```

Visual C#

```
bool FirstPagePrinted = false;
// Этот метод должен обработать событие PrintPage
public void PrintBigEllipse(object sender,
    System.Drawing.Printing.PrintPageEventArgs e)
{
    if (FirstPagePrinted == false)
    {
        FirstPagePrinted - true:
        e.HasMorePages - true;
        e.Graphics.DrawEllipse(Pens.Black, new Rectangle(0, 0,
            e.PageBounds.Width, e.PageBounds.Height * 2));
    }
    else
    {
        e.HasMorePages = false;
        FirstPagePrinted = false;
        e.Graphics.DrawEllipse(Pens.Black, new Rectangle(0,
            -(e.PageBounds.Height), e.PageBounds.Width,
            e.PageBounds.Height * 2));
    }
}
```

В этом примере управление вывода страниц на печать реализовано с помощью переменной *FirstPagePrinted*. Заметьте: эта переменная объявлена вне метода. Если объявить ее внутри метода, она инициализируется при каждом его вызове и всегда возвращает *false*. В действительности при печати каждой страницы программа перерисовывает эллипс заново, меняя его положение так, чтобы на текущую страницу попала соответствующая часть фигуры.

Печать текста

Вывод на печать текста происходит так же, как на экран, — с помощью метода *Graphics.DrawString*. Как и при выводе на экран, необходимо указать шрифт, сам текст, объект *Brush* и координаты начала печати, например:

Visual Basic .NET

```
Dim myfont As New Font( "Batang", 36. FontStyle.Regular, _
    GraphicsUnit.Pixel)
Dim Hello As String = "Hello World!"
e.Graphics.DrawString(Hello, myfont  Brushes.Black, 30. 30)
```

Visual C#

```
Font myfont = new Font("Batang", 36. FontStyle.Regular,
    GraphicsUnit.Pixel);
```

```
string Hello = "Hello World!";
e.Graphics.DrawString(Hello, myfont, Brushes.Black, 30, 30);
```

Примечание Перед печатью текста следует программно проверить, не выходит ли какая-то его часть за границы страницы: любое содержимое, которое выходит за границы страницы, напечатано не будет.

Печать многострочного текста

Для печати многострочного текста, например массива строк или содержимого текстового файла, необходим код, вычисляющий расстояние между отдельными строками. Чтобы рассчитать число строк на странице, разделите высоту клиентской области страницы (она равна высоте страницы за вычетом полей) на кегль шрифта. Положение любой строки на странице можно вычислить, умножив ее номер на кегль шрифта. Следующий пример демонстрирует печать строкового массива с именем myStrings:

Visual Basic .NET

```
' Эта переменная отслеживает текущий элемент массива,
' Во избежание инициализации этой переменной при печати каждой
' страницы, ее следует объявлять вне обработчика события PrintPage.
Dim ArrayCounter As Integer = 0
' Это обработчик события PrintDocument.PrintPage. Он предполагает,
' что строковый массив с именем myStrings() уже объявлен и заполнен,
' а также что инициализирован объект myFont, представляющий
' выбранный для печати шрифт.
Private Sub PrintStrings(sender As Object, e As _
   PrintPageEventArgs)
   ' Объявить переменные, управляющие межстрочными
   ' интервалами и разбиением документа на страницы.
   Dim LeftMargin As Single = e.MarginBounds.Left
   Dim TopMargin As Single = e.MarginBounds.Top
   Dim MyLines As Single = 0
   Dim YPosition As Single = 0
   Dim Counter As Integer = 0
   Dim CurrentLine As String
   ' Подсчитать число строк на странице,
   MyLines = e.MarginBounds.Height / _
      myFont.GetHeight(e.Graphics)
   ' Распечатать все строки файла, останавливаясь в конце страниц.
   While Counter < MyLines And ArrayCounter <= _
      myStrings.GetUpperBound(0)
      CurrentLine = myStrings(ArrayCounter)
      YPosition = TopMargin + Counter * _
         myFont.GetHeight(e.Graphics)
      e.Graphics.DrawString(CurrentLine, myFont, Brushes.Black, _
         LeftMargin, YPosition, New StringFormat())
```

```
        Counter += 1
        ArrayCounter += 1
    End While

    ' Если напечатаны еще не все строки, печатать следующую страницу.
    If Not (ArrayCounter = myStrings.GetUpperBound(0)) Then
        e.HasMorePages = True
    Else
        e.HasMorePages = False
    End If
End Sub
```

Visual C#

```
// Эта переменная отслеживает текущий элемент массива.
// Во избежание инициализации этой переменной при печати каждой
// страницы, следует объявлять ее вне обработчика события PrintPage.
int ArrayCounter = 0;
// Это обработчик события PrintDocument.PrintPage. Он предполагает,
// что строковый массив с именем myStrings() уже объявлен и заполнен,
// а также что инициализирован объект myFont, представляющий
// выбранный для печати шрифт.
private void PrintStrings(object sender, PrintPageEventArgs e)
{
    // Объявить переменные, управляющие межстрочными
    // интервалами и разбиением документа на страницы.
    float LeftMargin = e.MarginBounds.Left;
    float TopMargin = e.MarginBounds.Top;
    float MyLines = 0;
    float YPosition = 0;
    int Counter = 0;
    string CurrentLine;
    // Подсчитать число строк на странице.
    MyLines = e.MarginBounds.Height /
        myFont.GetHeight(e.Graphics);
    // Распечатать все строки файла, останавливаясь в конце страниц.
    while (Counter < MyLines && ArrayCounter <=
        myStrings.GetUpperBound(0))
    {
        CurrentLine = myStrings[ArrayCounter];
        YPosition = TopMargin + Counter *
            myFont.GetHeight(e.Graphics);
        e.Graphics.DrawString(CurrentLine, myFont, Brushes.Black,
            LeftMargin, YPosition, new StringFormat());
        Counter ++;
        ArrayCounter ++;
    }
```

```
    // Если напечатаны еще не все строки, печатать следующую страницу,
    if ( ! (ArrayCounter == myStrings.GetUpperBound(0)))
        e.HasMorePages = true;
    else
        e.HasMorePages = false;
}
```

Работа с цветами

Набор параметров для печати на принтере, поддерживающем цвета, отличается от того, что требуется для печати на черно-белом принтере. При цветной печати необходимо учесть, что некоторые цвета, хорошо различимые на экране, практически не видны на бумаге. Таким образом, код для печати на черно-белых и цветных принтерах не одинаков.

Чтобы определить, поддерживает ли текущий принтер цветную печать, проверьте свойство *PrinterSettings.SupportsColor*. Если этот так, свойство *DefaultPageSettings.Color* установлено в *true,* а принтер будет настроен на цветную печать по умолчанию. Для черно-белой печати установите свойство *DefaultPageSettings. Color* в *false.* Следующий пример демонстрирует организацию поддержки цветной и черно-белой печати с помощью свойства *PrinterSettings.SupportsColor.*

Visual Basic .NET

```
' Этот пример предполагает наличие операторов Irnports System.Drawing,
' System.Drawing.Drawing2D и System.Drawing.Printing
Dim BrushOne As Brush
Dim BrushTwo As Brush
If PrintDocument1.PrinterSettings.SupportsColor = True Then
    ' Создать кисти для цветной печати,
    BrushOne = Brushes.Red
    BrushTwo = Brushes.Blue
Else
    ' Создать объекты HatchedBrush для черно-белой печати.
    BrushOne = New HatchBrush(HatchStyle.DarkVertical, Color.Black)
    BrushTwo = New HatchBrush(HatchStyle.DashedHorizontal, Color.Black)
End If
```

Visual C#

```
// Этот пример предполагает наличие операторов Irnports System.Drawing,
// System.Drawing.Drawing2D и System.Drawing.Printing
Brush BrushOne;
Brush BrushTwo;
if (PrintDocument1.PrinterSettings.SupportsColor == true)
{
    // Создать кисти для цветной печати.
    BrushOne = Brushes.Red;
    BrushTwo = Brushes.Blue;
}
```

```
else
{
   // Создать объекты HatchedBrush для черно-белой печати.
   BrushOne = new HatchBrush(HatchStyle.DarkVertical, Color.Black);
   BrushTwo = new HatchBrush(HatchStyle.DashedHorizontal,
      Color.Black);
}
```

▶ **Организация поддержки печати в приложении**

1. Перетащите объект *Print Document* с панели Toolbox в окно дизайнера — в области компонентов появится новый экземпляр этого компонента, автоматически настроенный для использования принтера по умолчанию. Создать компонент *PrintDocument* можно и программно.
2. Напишите обработчик для события *PrintDocument.PrintPage.*
3. Добавьте к обработчику события *PrintPage* код, рисующий содержимое в клиентской области страницы. Передайте сформированное изображение на принтер с помощью объекта *PrintPageEventArgs.Graphics.*
4. Если вы работаете с многостраничными документами, напишите код, управляющий разбиением на страницы.

Применение элемента управления PrintPreviewControl

В .NET Framework имеется элемент управления *PrintPreviewControl,* обеспечивающий возможность предварительного просмотра документа перед отправкой его на печать. *PrintPreviewControl* находится на вкладке Windows Forms инструментальной панели Toolbox, откуда его можно перетащить на форму.

Чтобы настроить предварительный просмотр страницы, необходимо связать элемент управления *PrintPreviewControl* с экземпляром *PrintDocument,* установив свойство *PrintPreviewControl.Document* следующим образом:

Visual Basic .NET

```
myPrintPreview.Document = myPrintDocument
```

Visual C#

```
myPrintPreview.Document = myPrintDocument;
```

После этого элемент управления *PrintPreviewControl* будет отображать содержимое, предназначенное для печати. Вызовите обработчик события *PrintDocument.Print,* чтобы перехватить формируемый им графический образ страницы и отобразить его в элементе управления *PrintPreviewControl.*

При изменении параметров приложения часто меняется и облик печатаемого документа. Чтобы обновить образ страницы, отображаемый при предварительном просмотре, вызовите метод *InvalidatePreview:*

Visual Basic .NET

```
myPrintPreview.InvalidatePreview()
```

Visual C#

```
myPrintPreview.InvalidatePreview();
```

Zoom — еще одно важное свойство элемента управления *PrintPreviewControl*. Оно указывает масштаб страницы при предварительном просмотре. Если оно равно 1, масштаб равен 100 %. При значениях до 1 изображение для предварительного просмотра меньше оригинала, а при значениях, превышающих 1, соответственно больше.

Элемент управления PrintPreviewDialog

Элемент управления *PrintPreviewControl* существенно облегчает создание поддержки предварительного просмотра печатаемых документов, в том числе добавление нестандартных режимов просмотра. Однако большинству приложений вполне достаточно стандартных функций предварительного просмотра при печати. Для этих целей в .NET Framework предусмотрен стандартный элемент управления *PrintPreviewDialog*, поддерживающий самые распространенные возможности предварительного просмотра при печати. *PrintPreviewDialog* отображает диалоговое окно, показанное на рис. 8-1.

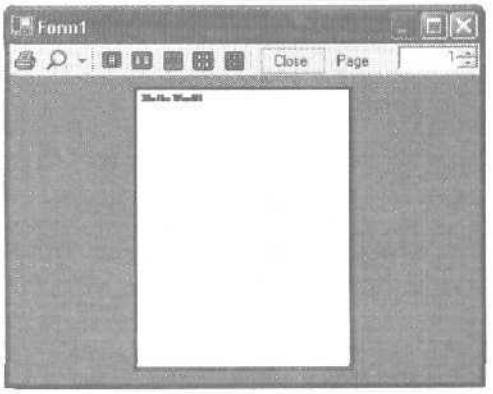


Рис. 8-1. Окно *PrintPreviewDialog*

Для предварительного просмотра документа следует поместить в свойство *Document* объекта *Print Preview Dialog* объект *Print Document*, представляющий нужный документ. Элемент управления *PrintPreviewDialog* позволяет просматривать несколько страниц документа одновременно, изменять масштаб и отправлять документ на печать. Вызов окна *PrintPreviewDialog* не отличается от вызова любого другого окна и осуществляется с помощью метода *Show* или *ShowDialog*.

Настройка параметров печати

Интерфейс печати .NET Framework поддерживает множество управляющих параметров. Свойство *Print Document Printer Settings* содержит сведения о принтерах, доступных в системе, свойство *PrintDocument.DefaultPageSettings* — параметры страницы по умолчанию (они применяются, если в обработчике события *PrintPage* не указаны другие параметры). В свою очередь, у этих свойств есть множество собственных свойств. К счастью, назначение большинства из них понятно без дополнительных пояснений. Кроме того, при создании объекта *PrintDocument* в свойство *PrinterSettings* помещается объект принтера по умолчанию, сконфигурированный параметрами по умолчанию, а в свойство *Default Page Settings* записываются параметры страницы по умолчанию. Таким образом, для создания и исполнения задания печати можно обойтись параметрами по умолчанию.

Если вы хотите разрешить пользователям самостоятельно конфигурировать задания печати, воспользуйтесь элементами управления Visual Studio .NET, обеспечивающими бесшовную интеграцию настройки принтера с пользовательским интерфейсом приложения — *PrintDialog* и *PageSetupDialog*.

PrintDialog

Окно *PrintDialog* позволяет установить свойство *PrinterSettings* объекта *PrintDocument* в период выполнения. Чтобы добавить экземпляр *PrintDialog* к приложению, перетащите его с вкладки Windows Forms панели Toolbox в окно конструктора. Чтобы отобразить окно *PrintDialog* в период выполнения, вызовите метод *ShowDialog* следующим образом:

Visual Basic .NET

```
PrintDialog1.ShowDialog()
```

Visual C#

```
PrintDialog1.ShowDialog();
```

Окно *PrintDialog* необходимо связать с объектом *PrintDocument*. Для этого поместите в свойство *Print Dialog. Document* объект *PrintDocument,* представляющий нужный документ.

Во время отображения окно *PrintDialog* (рис. 8-2) связывается со свойством *PrinterSettings* объекта *PrintDocument,* заданного свойством *Document,* и предоставляет пользователю графический интерфейс для настройки печати.

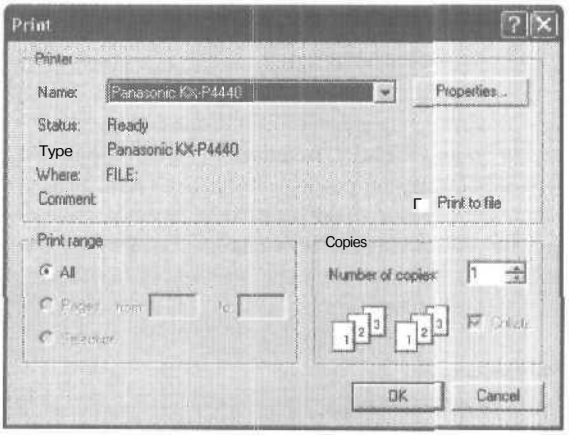


Рис. 8-2. Окно *PrintDialog*

PageSetupDialog

Методики применения окон *PageSetupDialog и PrintDialog* сходны. Окно *PageSetupDialog* (рис. 8-3) предоставляет графический интерфейс для настройки параметров страницы и печати в период выполнения. Пользователь может выбрать ориентацию страницы, размеры бумаги и установить поля, а также настроить принтер.

Как и *PrintDialog,* окно *PageSetupDialog* поддерживает свойство *Document,* которое содержит объект *PrintDocument,* представляющий нужный документ. Установив

свойство *Document,* можно связать объекты *PrintDocument* и *PageSetupDialog.* При этом любые изменения, сделанные в окне настройки печати, будут отражаться в свойствах объектов *PrintDocument. PrinterSettings* и *DefaultPageSettings.*

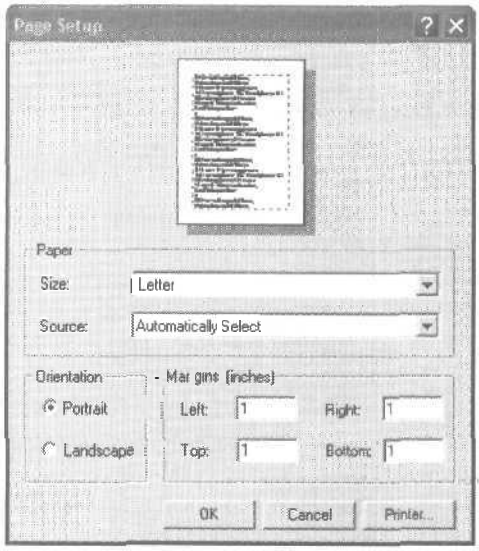


Рис. 8-3. Окно *PageSetupDialog*

Конфигурирование PageSettings в период выполнения

Иногда отдельные страницы многостраничного документа требуется напечатать не так, как остальные, например вывести одну страницу в альбомной ориентации, а все остальные — в книжной. Изменить параметры отдельной страницы позволяет свойство *PrintPageEventArgs.PageSettings,* которое предоставляет параметры печати страницы. Изменения параметров печати, сделанные с помощью этого свойства, действуют только на текущую страницу, остальные страницы задания печатаются, как задано при отправке документа на печать. Следующий пример демонстрирует изменение макета в период выполнения, устанавливая альбомную ориентацию страницы:

Visual Basic .NET

```
' Это фрагмент обработчика событий PrintPage.
' Переменная e представляет объект PrintPageEventArgs.
e.PageSettings.Landscape = True
```

Visual C#

```
// Это фрагмент обработчика событий PrintPage.
// Переменная e представляет объект PrintPageEventArgs.
e.PageSettings.Landscape = true;
```

Резюме

- Печатаемый документ представлен экземпляром класса *Print Document.* Этот класс поддерживает свойства *PrinterSettings*, задающее параметры принтера, и *DefaultPageSettings*,определяющее параметры страницы по умолчанию.
- Печать документа выполняют путем вызова метода *PrintDocument.Print.* Вызов этого метода генерирует событие *PrintPage*, *в* обработчик которого помещают код для прорисовки документа.
- Объект *PrintPageEventArgs* содержит все данные и поддерживает все функции, необходимые для прорисовки содержимого документа на принтере. Данные передаются на принтер в объекте *Graphics,* представляющем этот принтер; объект *Graphics* передается через объект *PrintPageEventArgs.*
- Для печати многостраничных документов к приложению необходимо добавить код, выполняющий разбиение документа на страницы. Для повторной генерации события *PrintPage* установите свойство *PrintEventArgs.HasMorePage в true.*
- Элемент управления *PrintPreviewControl* обеспечивает предварительный просмотр документов перед отправкой на печать, а *PrintPreviewDialog* предоставляет наиболее востребованные возможности *PrintPreview* в удобной для использования форме.
- Окна *PrintDialog* и *PageSetupDialog* дают пользователям возможность настраивать принтер и параметры страницы в период выполнения. Предусмотрена также возможность настройки параметров отдельной страницы — для этого достаточно изменить свойство *PrintPageEventArgs.PageSettings* во время печати.

Занятие 2. Получение доступа к компонентам и их вызов

Хотя в .NET Framework очень много встроенных возможностей, время от времени требуется доступ к внешним компонентам, таким, как собственная сборка, созданная с помощью .NET Framework, или существующий компонент COM. Также не исключены ситуации, когда необходимо вызывать Web-сервисы и напрямую взаимодействовать с Windows API. Visual Studio .NET позволяет задействовать внешние компоненты в приложения с затратой минимума усилий.

Изучив материал этого занятия, вы сможете:

- ✓ получать доступ к .NET-сборкам;
- ✓ получать доступ к компонентам COM и элементам управления ActiveX;
- ✓ вызывать Web-сервисы;
- ✓ вызывать функции Windows API.

Продолжительность занятия - около 30 **минут.**

Не подлежит сомнению, что рано или поздно вам понадобится доступ к внешнему коду. Неважно, с чем придется работать — с собственными компонентами, разработанными ранее, Web-сервисами или функциями Windows API, — средствами Visual Studio .NET вы без труда создадите ссылку на внешний компонент и сможете вызывать его как локальный код приложения.

Получение доступа к библиотекам типов .NET и COM

Средства .NET Framework позволяют легко получить доступ к любой .NET или COM-библиотеке, установленной в системе. В этих библиотеках могут быть компоненты собственных проектов, унаследованные компоненты COM или необходимая приложению бизнес-логика. Общая процедура получения доступа к компонентам .NET и COM предполагает создание ссылки на библиотеку типов. Для этого необходимо поместить в код приложения операторы *Import (use),* импортирующие библиотеку типов, и создать экземпляр нужного класса.

Список доступных библиотек типов отображается в диалоговом окне Add Reference (рис. 8-4). Чтобы вызвать его, щелкните правой кнопкой узел References в окне Solution Explorer (он располагается под узлом вашего проекта) и выберите из контекстного меню команду Add Reference.

В диалоговом окне Add Reference три вкладки. На вкладке .NET отображаются доступные .NET-сборки, на вкладке COM — список доступных библиотек типов COM, а на вкладке Projects — доступные проекты. Если элемента, на который вы хотите добавить ссылку, нет ни в одном из этих списков, щелкните кнопку Browse. Отыскав нужный файл, выберите его, щелкнув кнопку Select, — новая ссылка будет добавлена к проекту и появится в окне Solution Explorer.

Добавив ссылку на сборку или библиотеку типов COM, вы получаете возможность использовать в приложении типы из этой библиотеки, ссылаясь на них по полному имени. Чтобы упростить ссылки на члены из внешних библиотек, добавьте в код соответствующие операторы *Imports (using)* — о них рассказывается в главе 1. После этого разрешено объявлять типы из внешней библиотеки и создавать их экземпляры, как любые другие типы.

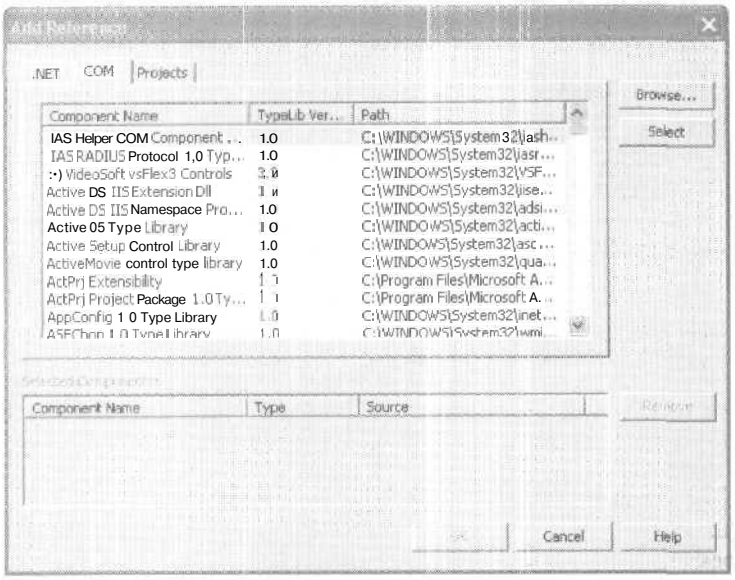


Рис. 8-4. Диалоговое окно Add Reference

▶ **Как получить доступ к .NET-сборке или библиотеке типов COM**

1. В окне Solution Explorer щелкните правой кнопкой узел References, расположенный под узлом вашего проекта, и из контекстного меню выберите команду Add Reference — откроется одноименное диалоговое окно.
2. Выберите объект, ссылку на который вы хотите добавить. Библиотеки типов COM перечислены на вкладке COM, .NET-сборки — на вкладке .NET, а проекты — на вкладке Projects.
3. Добавьте в код оператор *Imports* (*using*), чтобы импортировать внешний компонент в приложение.
4. После этого объявляйте импортированные типы и создавайте их экземпляры, как обычно.

Создание экземпляров элементов управления ActiveX

ActiveX — это одна из реализаций COM, поэтому процедура добавления ссылки на библиотеку типов элементов управления ActiveX практически не отличается от таковой для библиотеки COM. Выполнив ее, вы получаете возможность добавить элементы управления ActiveX на Toolbox, чтобы в период разработки добавлять их экземпляры к приложению.

Примечание Размещение элементов управления ActiveX на вкладке Windows Forms допустимо, хотя существенно снижает производительность. Поэтому элементы управления ActiveX следует использовать, только если нет .NET-элементов управления с соответствующими функциями.

▶ Создание экземпляра элемента управления ActiveX

1. Добавьте ссылку на библиотеку типов, в которой находится нужный элемент управления ActiveX (о добавлении ссылок рассказывалось немного раньше).
2. Поместите элемент управления ActiveX на панель Toolbox, как описано в главе 7.
3. Чтобы добавить к проекту экземпляр элемента управления в период разработки, перетащите его с Toolbox в окно дизайнера. Это можно сделать и по-другому, объявив класс элемента управления и создав его экземпляр в коде. Второй подход позволяет динамически создавать элементы управления ActiveX в период выполнения (глава 2).

Доступ к Web-сервисам

Web-сервисы (Web Service) — классы, размещенные в Интернете, — без сомнения относятся к важным компонентам .NET Framework. После объявления и создания экземпляра Web-сервиса в приложении вы сможете синхронно или асинхронно вызвать его методы.

Ссылки на Web-сервисы также создают в окне Solution Explorer. Для этого нужно вызвать это окно, щелкнуть правой кнопкой узел References, расположенный ниже узла вашего проекта, и выбрать из контекстного меню команду Add Web Reference — откроется одноименное окно (рис. 8-5), в котором задают URL Web-компонента, или находят его, если URL неизвестен.

В последнем случае вы получите доступ к каталогам, в которых хранится нужный ресурс. В каталоге Universal Description Discovery Integration (UDDI) хранится список провайдеров, которые предоставляют услуги в виде Web-сервисов. При выборе провайдера отображается список предоставляемых им Web-сервисов. Если щелкнуть Web-сервис, в левой панели появляется контакт XML, а в правой — имя Web-сервиса. Чтобы увидеть более «дружественное» описание Web-сервиса, удалите из URL строку «?WSDL» и перезагрузите страницу. Найдя нужную Web-ссылку, щелкните кнопку Add Reference, чтобы добавить ее в проект.

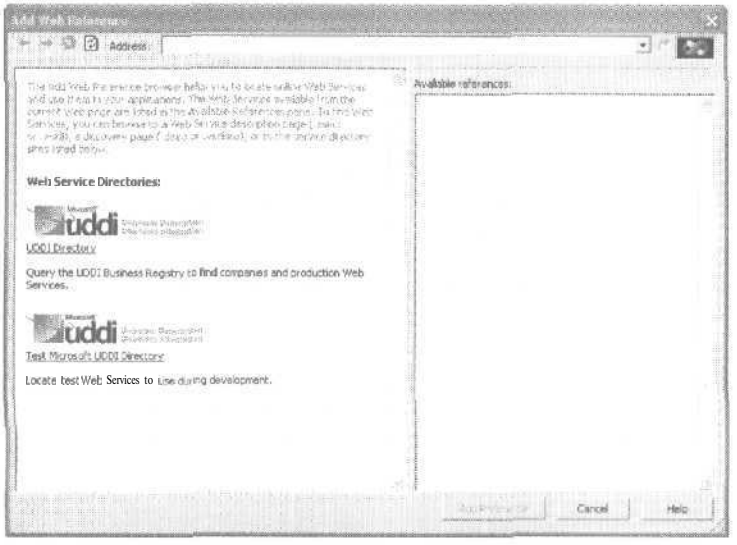


Рис. 8-5. **Окно Add Web Reference**

При создании экземпляра Web-сервиса .NET Framework в действительности создает экземпляр класса прокси, представляющего этот Web-сервис. Класс прокси существует в приложении и предоставляет все методы, поддерживаемые Web-сервисом. Вызов любого из этих методов передается Web-сервису, который находится в Интернете по заданному адресу. Экземпляры Web-сервисов создают так же, как экземпляры любых других компонентов. Следующий пример демонстрирует создание экземпляра Web-сервиса *WebService1*:

Visual Basic .NET

```
Dim myService As New WebService1
```

Visual C#

```
WebService1 myService - new WebService1();
```

Вызвать Web-сервис можно синхронно либо асинхронно. Синхронный вызов напоминает вызов обычной функции. В следующем примере показан синхронный вызов метода *myMethod* Web-сервиса *myService*. Этот пример предполагает наличие ссылки на экземпляр этого Web-сервиса:

Visual Basic .NET

```
myService.myMethod()
```

Visual C#

```
myService.myMethod();
```

Синхронный вызов метода Web-сервиса исполняется так же, как вызов обычного метода. Но, поскольку при этом происходит обращение к ресурсам Интернета, время отклика увеличивается, а исполнение приложения приостанавливается до завершения синхронного вызова Web-метода. Чтобы избежать задержек в работе программы из-за ожидания завершения вызова Web-сервиса, можно вызывать методы Web-сервиса асинхронно. В этом случае создается отдельный поток для взаимодействия с Web-сервисом, а пока Web-сервис обрабатывает запрос, исполнение программы продолжается, как обычно. На каждую синхронную версию метода Web-сервиса приходятся по две асинхронные, имена которых образуются из имени синхронной версии Web-метода и префиксов *Begin* и *End*. Так, если у Web-сервиса есть метод *MyMethod*, его асинхронные версии называются *BeginMyMethod* и *EndMyMethod*.

Асинхронный вызов Web-метода начинается с вызова соответствующего метода *Begin*. Помимо обычных параметров, требуемых при синхронном вызове, методу *Begin* необходим делегат *AsyncCallback*, представляющий метод обратного вызова для данного Web-метода, и параметр типа *Object*, позволяющий приложению передавать нестандартные сведения о состоянии.

Каждый метод *Begin* возвращает значение типа *IAsyncResult*. Этот интерфейс используется соответствующим методом *End* для извлечения результатов асинхронного метода. Сигнатура метода обратного вызова, заданного делегатом *AsyncCailback,* должна включать параметр типа *lAsyncResult*. В свою очередь, метод обратного вызова вызывает метод *End,* который извлекает данные. Следующий пример иллюстрирует асинхронный вызов метода *MyMethod* с указанием метода обратного вызова и извлечение результатов *MyMethod*. При этом предполагается, что *MyMethod* — это метод Web-сервиса с именем *WebServicel,* возвращающий строковое значение.

Visual Basic .NET

```
Public Class AsyncDemo
   Dim myService As WebService1
   Public Sub CallMethodAsynchronously()
      myService - New WebService1 ()
      ' Создать делегат AsyncCallback с помощью оператора AddressOf.
      ' Этот объект необходим для вызова метода, но в этом примере
      ' он не используется.
      myService.BeginMyMethod(AddressOf CallBack, New Object())
   End Sub
   Public Sub CallBack(ByVal e as lAsyncResult)
      Dim myString As String
      ' Данные извлекают с помощью метода 'End', передав ему
      ' в качестве параметра объект IAsyncResult.
      myString = myService.EndMyMethod(e)
   End Sub
End Class
```

Visual C#

```
public class AsyncDemo
{
   WebService1 myService;
   public void CallMethodAsynchronously()
   {
      myService = new WebService1();
      // Создать делегат AsyncCallback, указывающий метод обратного вызова.
      System.AsyncCallback myCallBack - new
         System.AsyncCallback(CallBack);
      // Этот объект необходим для вызова метода, но
      // в этом примере он не используется.
      myService.BeginMyMethod(myCallBack, new object());
   }
   public void CallBack(IAsyncResult e)
   {
      string myString;
      // Данные извлекают с помощью метода 'End', передав ему
      // в качестве параметра объект IAsyncResult.
      myString = myService.EndMyMethod(e);
   }
}
```

Указать метод обратного вызова необходимо, но использовать его не обязательно, поскольку можно извлечь данные в методе *Begin*, вызвав *End* напрямую. Если метод *End* вызывается асинхронно, исполнение программы приостанавливается до его завершения. Вот пример, иллюстрирующий это:

Visual Basic .NET

```
Public Sub AsyncDemo()
   Dim myService As New WebService1
   Dim Async as IAsyncResult
   ' Записать IasyncResult, который вернул метод 'Begin' в локальную
   ' переменную. Делегат метода обратного вызова необходим,
   ' но для извлечения данных он не используется.
   Async - myService.BeginMyMethod(AddressOf SomeMethod, New Object())
   ' Здесь выполняются действия, сильно загружающие процессор.
   Dim myString As String
   ' Если метод Begin еще не завершился, исполнение приложения
   ' приостанавливается, пока это не произойдет.
   myString = myService.EndMyMethod(Async)
End Sub
```

Visual C#

```
publicvoidAsyncDemoO
{
   WebService1 myService = new WebService1();
   IAsyncResult Async;
   // Записать IasyncResult, который вернул метод 'Begin' в локальную
   // переменную. Делегат метода обратного вызова необходим,
   // но для извлечения данных он не используется.
   System.AsyncCallback myCallBack - new
      System.AsyncCallback(SomeMethod);
   Async = myService.BeginMyMethod(myCallBack, new object());
   // Здесь выполняются действия, сильно загружающие процессор,
   string myString;
   /,/ Если метод Begin еще не завершился, исполнение приложения
   // приостанавливается, пока это не произойдет.
   myString = myService.EndMyMethod(Async);
}
```

▶ **Создание ссылки на Web-сервис**

1. В окне Solution Explorer щелкните правой кнопкой узел References и выберите из контекстного меню команду Add Web Reference.
2. Укажите ссылку на Web-сервис, найденный в каталоге UDDI, либо введите адрес Web-сервиса вручную. Щелкните кнопку Add Reference, чтобы добавить ссылку в проект.
3. Добавив ссылку, можно создавать экземпляры Web-сервиса, как обычно.

▶ **Как синхронно вызвать метод Web-сервиса**

1. Создайте экземпляр Web-сервиса.
2. Вызовите нужный метод, как обычно.

▶ **Как асинхронно вызвать метол Web-сервиса**

1. Объявите метод обратного вызова, принимающий параметр типа *IAsyncResult.*
2. Инициируйте асинхронный вызов. Для этого вызовите метод *Begin* соответствующего Web-метода, передав ему делегат метода обратного вызова.
3. Из метода обратного вызова (или другого метода) вызовите соответствующий метод *End,* чтобы извлечь данные; передайте ему объект *IAsyncResult,* который вернул метод *Begin.*

Вызов функций Windows API

После появления .NET Framework возможность напрямую работать с встроенными функциями операционной системы уже не так важна, как раньше. Для ключевых функций Windows API созданы оболочки в виде .NET-классов, обеспечивающие контроль типов и исключающие ошибки при вызове функций ОС.

Тем не менее время от времени вызывать неуправляемые функции все же приходится. Для импорта встроенных функций ОС служит ключевое слово *Declare* (в Visual Basic .NET) и атрибут *DHImportAttribute* (в Visual C#). Импортированную функцию можно использовать в своем приложении, как обычную. Например, объявить функцию Windows API *Beep*:

Visual Basic .NET

```
Private Declare Function Beep Lib "kernel32" (ByVal dwFreq As _
   Integer, ByVal dwDuration As Integer) As Integer
```

Visual C#

```
[System.Runtime.InteropServices.DllImport("kernel32")]
private static extern int Beep(int dwFreq, int dwDuration);
```

Это объявление включает модификатор доступа *(Private* или *private),* определяющий уровень доступа к этой функции в приложении. В Visual Basic .NET внешнюю функцию указывают с помощью ключевого слова *Declare,* а в Visual C# — с помощью ключевых слов *static* и *extern* (для объявления функций API необходимы оба этих слова). В Visual C# любую внешнюю функцию необходимо предварять атрибутом *DllImportAttribute,* указывающим имя библиотеки, в которой находится импортируемая функция. В Visual Basic .NET имя DLL задают в виде строки после ключевого слова *Lib.* Объявление также включает имя функции (в данном случае *Beep),* под которым она существует в DLL (это имя чувствительно к регистру). Кроме того, сигнатура и тип возвращаемого значения у внешней и внутренней функций должны быть идентичными.

Примечание В Visual Basic .NET имена многих типов изменились по сравнению с Visual Basic 6 (например, прежний тип *Long* теперь стал *Integer)* — не забывайте об этом при объявлении внешних функций.

Внешнюю функцию разрешается вызвать, как обычную функцию.

▶ **Как объявить внешнюю функцию на Visual Basic .NET**

Для этого применяют ключевое слово *Declare.* Ключевое слово *Lib* указывает DLL, из которой импортируют функцию, имя и сигнатура этой функции должны быть полностью идентичными таковым внешней функции.

Как объявить внешнюю функцию на Visual C#

Воспользуйтесь атрибутом *DllImportAttribute* из пространства имен *System.Runtime.InteropServices*, чтобы указать DLL, откуда импортируется функция. В объявлении функции необходимы ключевые слова *static* и *extern,* а ее имя и сигнатура должны быть идентичными таковым внешней функции.

Резюме

- Чтобы задействовать в своем приложении .NET-сборку или библиотеку типов COM, добавьте ссылку на эту библиотеку типов и создайте экземпляр нужного компонента. Аналогично поступают с элементами управления ActiveX, их к тому же разрешается добавлять на панель Toolbox.
- Чтобы получить доступ к Web-сервису, необходимо добавить к приложению Web-ссылку. После этого в коде можно объявить Web Service и создать его экземпляр. В результате создается класс-оболочка, предоставляющая методы Web-сервиса.
- Синхронный вызов методаWeb-сервиса не отличается от вызова обычных методов. До завершения синхронно вызванного метода исполнение приложения приостанавливается.
- Асинхронный вызов выполняется в два этапа. Сначала вызывают метод *Begin.* Ему необходим делегат метода, принимающего параметр типа *lasyncResult.* Далее вызывают метод *End* и передают ему объект *lasyncResult,* возвращенный методом *Begin*, — из него *End* извлекает данные, которые и являются результатом асинхронного метода.
- Для объявления внешних функций служит ключевое слово *Declare* (вVisual Basic .NET) или *static* и *extern* (в Visual C#). Также необходимо указать имя библиотеки, откуда импортируют функцию. Это делается с помощью ключевого слова *Lib* (в Visual Basic .NET) или атрибута *DllImportAttribute* (в Visual C#). Имя и сигнатура функции должны быть идентичны таковым внешней функции.

Занятие 3. Реализация дополнительных возможностей доступа

При разработке приложений следует подумать обо всех потенциальных пользователях. Среди них могут быть люди, испытывающие трудности при работе со стандартным пользовательским интерфейсом, реализованным в большинстве приложений Windows Forms. Встраивая в свои приложения специальные возможности доступа, удовлетворяющие нужды таких людей, вы расширите круг пользователей своих программ и сделаете их более удобными.

Изучив материал этого занятия, вы сможете:

- ✓ изложить принципы разработки приложений, поддерживающих специальные возможности;
- ✓ описать требования к реализации специальных возможностей, необходимые для получения логотипа «Certified for Windows»;
- ✓ рассказать о средствах поддержке специальных возможностей стандартными элементами управлениями Windows Forms и описать назначение этих средств.

Продолжительность занятия - **около 20** минут.

Некоторым пользователям для доступа к приложениям необходимы специальные возможности. Согласно современным требованиям к разработке программного обеспечения для бизнеса, приложения следует делать доступными людям с разным уровнем физических возможностей.

Есть несколько подходов к реализации этого требования. В Windows XP имеются служебные программы, реализующие специальные возможности доступа в ваших собственных приложениях. Например, технология SoundSentry позволяет сопровождать звуковые сигналы визуальным оповещением. Все эти средства поддерживают бесшовную интеграцию с приложениями, в которых вы хотите реализовать специальные возможности. При этом необходимо соблюдать ряд принципов.

Реализация специальных возможностей на этапе проектирования

Заблаговременное планирование облегчает создание приложений, пользовательский интерфейс которых соответствует принципам разработки с учетом различных физических возможностей пользователей. Вот какие принципы следует заложить в приложение еще на этапе его проектирования:

- гибкость;
- наличие нескольких методов ввода и вывода данных;
- логичную и согласованную структуру интерфейса;
- совместимость со средствами, обеспечивающими специальные возможности.

Ключевым принципом проектирования специальных возможностей считается гибкость. Пользователям требуется возможность настройки интерфейса «под себя», а также возможность выбора способа ввода данных (посредством указывающих устройств или с клавиатуры). Все ключевые функции приложения должны быть дос-

тупными как с клавиатуры, так и с помощью мыши. Пользователи также хотят самостоятельно выбирать метод вывода информации; звуковой, визуальный (графический и текстовый). Функциональность вашего приложения и реализация его взаимодействия с другими приложениями должны быть согласованными и подчиняться единому стилю. Наконец, приложение должно поддерживать средства, реализующие специальные возможности.

Специальные возможности и логотип «Certified for Windows»

Стремясь к стандартизации правил разработки приложений, доступных пользователям с различными возможностями, Microsoft выдвинула пять требований, на соответствие которым приложения проверяются при сертификации в рамках программы «Certified for Windows». В этом разделе описаны требования программы «Certified for Windows» и способы их реализации.

Поддержка стандартных параметров

Первое требование гласит, что приложения должны поддерживать стандартные параметры, управляющие размером и цветами элементов интерфейса, шрифтами и способами ввода информации. Это позволит обеспечить согласованность пользовательского интерфейса любых приложений и системы. Таким образом, пользователи, нуждающиеся в специальных возможностях доступа, смогут централизованно настроить параметры интерфейса, причем последние автоматически будут применены ко всем установленным в системе приложениям.

Данное требование реализуется посредством объектов из поддерживаемого .NET Framework пространства имен *System*. .NET Framework предоставляет доступ к системной палитре для оформления элементов интерфейса приложения. В результате изменения общесистемных параметров цветового оформления интерфейса автоматически применяются к интерфейсу вашего приложения. Управление шрифтами также можно оставить на усмотрение системы, а не устанавливать свойство *Font* явно для элементов управления, составляющих интерфейс приложения. Если свойству *Font* оставлено значение по умолчанию, оно устанавливается в соответствии с системными параметрами. Исключением является свойство *Font.Size*, которое можно менять, не нарушая требования к реализации специальных возможностей доступа. Приложение следует спроектировать так, чтобы с ним было легко работать при разрешении экрана 640x480. Для ввода данных в приложение должно быть достаточно возможностей клавиатуры и указывающих устройств.

Поддержка высококонтрастных цветовых схем

Если пользователям необходимо выводить текст и графику с максимальной четкостью, им предлагаются высококонтрастные цветовые схемы (параметр High Contrast), обеспечивающих максимальный контраст элементов пользовательского интерфейса, окрашенных в светлые и темные тона.

Чтобы гарантировать поддержку приложением высококонтрастных цветовых схем, следует применять только системные цвета либо цвета, выбранные пользователем. В случае активизации высококонтрастной цветовой схемы интерфейс вашего приложения автоматически приводится в соответствие с выбранной схемой. Следует избегать применения фоновых изображений с формами, поскольку они снижают контрастность и затрудняют восприятие содержимого.

Наличие документированных возможностей доступа с клавиатуры ко всем функциям программы

В приложении необходимо реализовать доступ с клавиатуры ко всем функциям. При этом все клавиши и их сочетания, необходимые для доступа к тем или иным функциям, должны быть указаны в легко доступной документации. Следует настроить быстрые клавиши для доступа к элементам управления и меню, а также ясные правила перехода по клавише Tab. Нельзя недооценить важность правильного отражения этих возможностей в документации. Следует предоставить пользователю простую возможность выяснить, какие клавиши для каких целей используются в приложении. Ничто так не способствует этому, как тщательно составленная, полная и рационально организованная документация.

Поддержка следования за фокусом ввода

Сведения о фокусе ввода с клавиатуры используются такими программами, как Экранная лупа (Magnifier) и Экранный диктор (Narrator), поэтому важно, чтобы и пользователь, и приложение всегда четко знали, где находится фокус. В большинстве случаев эту проблему удается решить средствами .NET Framework. Тем не менее при проектировании приложения необходимо предусмотреть код, который при первом отображении формы устанавливает фокус на первый элемент управления, и определить порядок перехода фокуса по клавише Tab в соответствии с последовательностью действий при работе с формой.

Использование различных средств отображения информации

Приложение не должно полагаться только на звуковые сигналы при оповещении пользователя о важных событиях. Для одних пользователей звуковые сигналы имеют первостепенное значение, а другие могут просто не услышать их. Поэтому при передаче информации следует комбинировать звуковые сигналы с визуальными оповещениями, такими, как мерцание фона в активном окне, вывод сообщений и применение немодального экрана, отображающего сведения во время работы приложения.

Специальные возможности и элементы управления Windows Forms

Помимо свойств, управляющих обликом интерфейса, каждый элемент управления Windows Forms поддерживает пять свойств (таблица 8-2), контролирующих специальные возможности доступа и взаимодействие элемента управления с реализующими их программами.

Таблица 8-2. Свойства элементов управления Windows Forms, обеспечивающие поддержку специальных возможностей доступа

Имя	Описание
AccessibleDescription	Содержит описание для программ, реализующих специальные возможности
AccessibleName	Содержит имя, которое используется служебными программами, реализующими специальные возможности

Таблица 8-2. *(окончание)*

Имя	Описание
AccessibleRole	Содержит описание функций элемента управления, которые используются средствами, реализующими специальные возможности
AccessibilityObject	Содержит экземпляр *AccessibleObject*, предоставляющий информацию об элементе управления программам, которые реализуют специальные возможности. Это свойство доступно только для чтения и устанавливается дизайнером
AccessibleDefaultActionDescription	Содержит описание действия по умолчанию для элемента управления. Это свойство нельзя установить в период разработки — только программно

Эти свойства предоставляют служебным программам, реализующим специальные возможности, различные сведения о роли элементов управления в приложении. Программы затем представляют собранную ими информацию пользователю или применяют ее для определения способа отображения элементов управления.

Свойство *AccessibilityObject* содержит экземпляр класса *AccessibleObject*, предоставляющий информацию соответствующим служебным программам. Сведения, предоставленные свойствами *AccessibleName*, *AccessibleDescription* и *AccessibleDefaultActionDescription*, позволяют служебным программам, которые обеспечивают специальные возможности доступа, описать пользовательский интерфейс вашего приложения. Свойство *AccessibleRole* содержит перечислимое значение, определяющее способ взаимодействия служебных программ с данным элементом управления. Обычно оно установлено в *Default*. Это означает, что каждый элемент управления функционирует в обычном режиме. Если же требуется более согласованное взаимодействие с элементами управления других типов, а также с нестандартными элементами управления, следует присвоить этому свойству соответствующее значение. Например, если на форме имеется нестандартный элемент управления, отображающий данные в виде диаграммы, свойство *AccessibleRole* надо установить в *Chart*, что позволит служебным программам обращаться с ним, как с диаграммой.

Резюме

- Чтобы заложить основы для реализации специальных возможностей, при проектировании приложения необходимо соблюдать следующие принципы:
  - гибкость;
  - реализацию разнообразных методов для ввода и вывода информации;
  - согласованность и упорядоченность интерфейса и функциональности;
  - поддержку служебных программ, обеспечивающих специальные возможности.
- К программам–кандидатам на получение логотипа «Certified for Windows» предъявляется ряд требований, связанных со специальными возможностями. Приложение должно поддерживать:
  - стандартные системные параметры, управляющие оформлением интерфейса;
  - высококонтрастные цветовые схемы:

- доступ с клавиатуры к любым элементам пользовательского интерфейса, Все задействованные клавиши и их сочетания должны быть описаны в документации;
- уведомление о положении фокуса ввода;
- дополнительные средства уведомления о событиях помимо звуковых сигналов,

- Элементы управления Windows Forms поддерживают пять свойств, связанных с реализаций специальных возможностей доступа;
  - *Accessible Description*
  - *Accessible Name*
  - *Accessible Role*
  - *Accessibility Object*
  - *Accessible Default Action Description*

Занятие 4. Создание справочной системы

Значение качественной документации на программное обеспечение слишком часто недооценивают. При наличии справочной системы пользователи быстрее осваивают ПО, что повышает производительность их труда и позволяет снизить затраты на обучение. На этом занятии вы узнаете, как создать справочную систему для приложения.

Изучив материал этого занятия, вы сможете:

- ✓ отображать справочные сведения с помощью класса *Help*;
- ✓ применять компонент *HelpProvider*.

Продолжительность занятия - около 20 минут.

Сложные приложения непременно следует снабдить встроенной справочной системой. Это может быть набор связанных HTML-файлов, представляющих разделы справочной системы, либо набор CHM-файлов (Compressed HTML), созданных с помощью HTML Help Workshop. Оба варианта позволяют динамически отображать связанные разделы справочной системы.

Работа с классом Help

Класс *Help* позволяет приложению отображать пользователям файлы справочной системы в формате HTM или CHM. Класс *Help* инкапсулирует ядро HTML Help 1.0 и поддерживает два статических метода для отображения справочных файлов – *ShowHelp* и *ShowHelp Index*.

Первый требуется для отображения файлов со справочными сведениями для определенного элемента управления. Для вызова этого метода необходима ссылка на родительский элемент управления окна *HelpDialog*, а также URL справочного файла. Это может быть URL локального файла (например, C:\myHelp.htm) или HTTP (HTTP://myHelp.htm). Поскольку метод *ShowHeip* является статическим, создавать экземпляр класса *Help* перед вызовом этого метода не требуется, тем более что создавать экземпляры этого класса не разрешается. Вот как отобразить справочный файл с помощью метода *ShowHeip:*

Visual Basic .NET

```
Help.ShowHelp(MyForm, "C:\myHelpFile.htm")
```

Visual C#

```
Help.ShowHelp(MyForm, @"C:\myHelpFile.htm");
```

Этому методу можно также передать параметр *HelpNavigator*. Он определяет отображаемые элементы справочного файла и принимает значение *TableOfContents*, *Find, Index* или *Topic,* Также разрешается задавать ключевые слова для поиска:

Visual Basic .NET

```
Help.ShowHelp(MyForm, "C:\myHelpFile.htm", "HelpMenu")
```

Visual C#

```
Help.ShowHelp(MyForm, @"C:\myHelpFile.htm", "HelpMenu");
```

Метод *ShowHelpIndex* класса *Help* применяется для отображения предметного указателя для заданного справочного файла. Этот метод вызывают так же, как *ShowHelp*:

Visual Basic .NET

```
Help.ShowHelpIndex(MyForm, "C:\myHelpFile.htm")
```

Visual C#

```
Help.ShowHelpIndex(MyForm, @"C:\myHelpFile.htm");
```

Методы класса *Help* можно вызвать в ответ на самые разные события, генерируемые элементами интерфейса. Например, у многих приложений в меню есть пункт Help; одноименный пункт часто включают в контекстное меню элементов управления, чтобы предоставить доступ к соответствующим разделам справочной системы. Проектируя приложение, необходимо предусмотреть возможность доступа к справочной системе там, где это логично и удобно для пользователя.

Применение компонента HelpProvider

Компонент *HelpProvider* позволяет снабдить элементы управления пользовательского интерфейса справочными описаниями. *HelpProvider* представляет собой провайдер дополнительных свойств, координирующий и поддерживающий их для всех элементов управления, добавляемых к форме. В свойство *HelpString любого* элемента управления форме можно поместить строку, которая будет отображаться при получении элементом управления фокуса либо по нажатию F1, а в свойство *HelpNameSpace* поместить URL справочного файла, связанного с компонентом *HelpProvider*.

Компонент *HelpProvider* предоставляет каждому размещенному на форме элементу управления три свойства:

- *HelpString*
- *HelpKeyWord*
- *HelpNavigator*

Для любого элемента управления эти свойства разрешается устанавливать в период разработки через окно Properties или программно:

Visual Basic .NET

```
myHelpProvider.SetHelpString(Button1, _
"This Button initiates the self-destruct sequence")
```

Visual C#

```
myHelpProvider.SetHelpString(Button1,
"This Button initiates the self-destruct sequence");
```

Если свойство *HelpNameSpace* не установлено, автоматически отображаются только строки из *HelpString* в виде всплывающих подсказок, а остальные свойства игнорируются. Если же свойство *HelpNameSpace* определяет справочный файл, то при вызове справки отображается содержимое этого файла с использованием параметров, заданных свойствами *HelpNavigator* и *HelpKeyword*. Свойство *HelpNavigator* принимает одно из следующих значений:

- *TableOfContents* — показывает оглавление справочной системы;
- *Find* — показывает страницу поиска;
- *Index* — выводит предметный указатель;
- *Topic* — выводит заданный раздел справочной системы;
- *Associated Index* — отображает предметный указатель заданного раздела;
- *KeywordIndex* — выводит результаты поиска по ключевому слову.

Если свойство *HelpNameSpace* установлено, строка из *HelpString* по нажатию F1 не отображается, но остается доступной. Чтобы получить значение *HelpString* для некоторого элемента управления, вызовите метод *HelpProvider.GetHelpString*:

Visual Basic .NET

```
myHelpProvider.GetHelpString(Button1)
```

Visual C#

```
myHelpProvider.GetHelpString(Button1);
```

Резюме

- Класс *Help* поддерживает статические методы для отображения содержимого справочной системы приложения. Для отображения справочных сведений предусмотрен метод *Help.ShowHelp*. Приложения следует проектировать так, чтобы при необходимости пользователи легко получали доступ к справочной системе.
- Компонент *HelpProvider* позволяет отобразить либо значение *HelpString*, либо раздел справки по нажатию F1. Разрешается устанавливать отдельные свойства *HelpString*, *HelpKeyWord* и *HelpNavigator* для каждого элемента управления, размещенного на форме. Если свойство *HelpNameSpace* не установлено, значения свойства *HelpString* отображаются как всплывающие подсказки. Если свойство *HelpNameSpace* задано, то вместо всплывающих подсказок отображаются соответствующие разделы справочной системы.

Занятие 5. Глобализация и локализация

.NET Framework предоставляет уникальные возможности для разработки приложений, предназначенных пользователям разных стран. Ваши приложения будут поддерживать различные языки, форматы денежных единиц, даты, времени и др. значений, формат которых зависит от культуры, к которой принадлежит пользователь, На этом занятии изложены методики глобализации и локализации приложений.

Изучив материал этого занятия, вы сможете:

- ✓ дать определение понятиям «глобализация» и «локализация»;
- ✓ реализовать глобализацию и локализацию пользовательского интерфейса;
- ✓ подготовить сведения о форматировании для конкретной культуры;
- ✓ задавать отображение текста справа налево;
- ✓ преобразовывать тексты в различных кодировках;
- ✓ проверять правильность ввода данных на языках, не использующих латинский алфавит.

Продолжительность занятия - около 30 минут.

Если ваша компания работает на международном рынке, ей понадобится программное обеспечение, с которым смогут работать пользователи, принадлежащие к различным культурам. Некоторые пользователи не знают ни стандартов форматирования денежных единиц и дат, принятых в США, ни английского языка. Снабдив приложение поддержкой нескольких культур, вы расширите область его применения и подготовите его к выходу на международный рынок.

Определение глобализации и локализации

Терминами «глобализация» и «локализация» называют два разных, но взаимосвязанных процесса. Под глобализацией понимают применение к существующим данным формата, зависимого от культуры, а под локализацией — извлечение данных, соответствующих заданной культуре. Вот примеры, иллюстрирующие эти различия.

- Глобализация. В некоторых странах при форматировании значений денежных единиц точкой разделяют разряды тысяч, а запятой отделяют целую часть от дробной в десятичных дробях. В приложении, реализующем глобализацию, значения денежных сумм форматируются согласно стандартам региона, в котором оно используется.
- Локализация. В зависимости от страны, в которой используется приложение, название формы можно отображать по-английски или по-французски. Локализованное приложение извлекает и отображает строку на соответствующем языке в зависимости от региона, в котором оно используется.

Концепция культур

Термином «культура» при разработке приложений называется совокупность региональных стандартов, принятых в стране, где используется приложение. В .NET Framework культура определяется специальным значением, которое называется *кодом культуры*. Код культуры — это значение, описывающее для инфраструктуры применяемый в данный момент язык. В код культуры дополнительно можно вклю-

чить сведения о регионе, где приложение используется. Обычно код культуры состоит из одной пары букв — в этом случае он определяет только язык — либо комбинации из двух пар букв (кодов языка и региона соответственно), разделенных дефисом. Культуры, заданные двузначными кодами и определяющие только язык, называются *нейтральными,* а культуры, заданные четырехзначным кодом и определяющие язык и регион, называются *специфичными.* Вот несколько примеров кодов культур:

- En — английский язык без указания региона;
- **en-CA** — канадский вариант английского языка;
- **af-ZA** — африкаанс (Южная Африка);
- **eu** — баскский язык без указания региона;
- **kn-IN** — язык каннада (Индия);
- **tr** — турецкий язык без указания региона.

Полный список кодов культур вы найдете в разделе справочной системы .NET Framework, посвященному классу *CultureInfo.*

Коды большинства культур подчиняются такому формату, но есть и исключения, о которых следует помнить:

- **Cy-uz-UZ** — узбекский язык (Узбекистан) с кириллическим алфавитом;
- **Lt-uz-UZ** — узбекский язык (Узбекистан) с латинским алфавитом;
- **zh-CHT** — традиционное китайское письмо без указания региона;
- **zh-CHS** — упрощенное китайское письмо без указания региона.

Получение **и установка текущей** культуры

Приложение автоматически считывает код культуры, заданный системными параметрами, и использует соответствующую культуру. Чтобы программно изменить текущую культуру приложения, запишите в соответствующее свойство экземпляр класса *CultureInfo,* представляющий новую культуру. Этот класс содержит сведения о культуре и соответствующие инструкции для приложения, например информацию о календаре, формат денежных единиц, дат и т. д. Чтобы программно назначить культуру приложению, поместите в свойство *CurrentThread.CurrentCulture* новый экземпляр класса *CultureInfo.* Класс *CultureInfo* воспринимает в качестве параметра строку, представляющую код соответствующей культуры. Следующий пример демонстрирует способ замены текущей культуры франко-канадской:

Visual Basic .NET

```
System.Threading.Thread.CurrentThread.CurrentCulture = New _
   System.Globalization.CultureInfo("fr-CA")
```

Visual C#

```
System.Threading.Thread.CurrentThread.CurrentCulture = new
   System.Globalization.CultureInfo("fr-CA");
```

Чтобы извлечь класс *CultureInfo,* представляющий текущую культуру, вызовите класс *CultureInfo. CurrentCulture* следующим образом:

Visual Basic .NET

```
' Этот пример предполагает наличие оператора Imports System.Globalization
Dim myCurrentCulture As CultureInfo
myCurrentCulture = CultureInfo.CurrentCulture
```

Visual C#

```
// Этот пример предполагает наличие оператора using System.Globalization
CultureInfo myCurrentCulture;
myCurrentCulture=CultureInfo.CurrentCulture;
```

Глобализация приложений

Если записать в свойство *Thread.CurrentThread.CurrentCulture* новый экземпляр класса *CultureInfo,* все данные, форматируемые приложением, автоматически обновятся согласно правилам, принятым в установленной культуре, а данные, за форматирование которых приложение не отвечает, останутся без изменений. Рассмотрим форму с единственным элементом управления Label, для которого программно задан следующий текст:

Visual Basic .NET

```
Label1.Text = "$500.00"
```

Visual C#

```
Label1.Text = "$500.00";
```

После этого текущую культуру установили в en-GB (Великобритания). Как вы думаете, какое значение отобразит надпись? Он останется прежним — $500.00. Смена культуры не повлияет на содержимое надписи, поскольку оно не отформатировано. Если же задать значение элемента управления Label следующим образом:

Visual Basic .NET

```
Label1.Text = Format(500, "Currency")
```

Visual C#

```
Label1.Text = (500).ToString("C");
```

то при смене культуры на en-GB значение надписи изменится на £500.00, поскольку к нему применен формат денежных единиц, принятый в новой культуре. Обратите внимание, что сумма осталась прежней — изменился только формат.

Локализация приложений

Локализация позволяет изменять вид пользовательского интерфейса в зависимости от культуры. .NET Framework облегчает создание локализованных форм благодаря использованию файлов ресурсов. Эти файлы хранят данные для альтернативных версий форм, соответствующих различным культурам, поддерживаемым приложением. Версия формы, загружаемая в период выполнения, определяется свойством *CultureInfo. CurrentUICulture.*

Получение и установка текущей культуры пользовательского интерфейса

Культура, назначенная пользовательскому интерфейсу, также представлена экземпляром класса *CultureInfo*, и может отличаться от культуры, заданной свойством *CultureInfo.CurrentCulture*. Свойство *CurrentCulture* определяет правила форматирования данных, а свойства *CurrentUICulture* — версию ресурсов, загружаемых в локализованные формы в период выполнения. Получить значение свойства *CurrentUICulture* можно так:

Visual Basic .NET

```
' Этот пример предполагает наличие оператора Imports System.Globalization
Dim myCurrentCulture As CultureInfo
myCurrentCulture = CultureInfo.CurrentUICulture
```

Visual C#

```
// Этот пример предполагает наличие оператора using System.Globalization
CultureInfo myCurrentCulture;
myCurrentCulture = CultureInfo.CurrentUICulture;
```

Пользовательскому интерфейсу назначают текущую культуру так же, как и приложению, — через объект *Thread*, представляющий текущий поток. Например, именно так можно назначить интерфейсу Тайскую культуру:

Visual Basic .NET

```
System.Threading.Thread.CurrentThread.CurrentUICulture = New _
   System.Globalization.CultureInfo("th-TH")
```

Visual C#

```
System.Threading.Thread.CurrentThread.CurrentUICulture = new
   System.Globalization.CultureInfo("th-TH");
```

После установки текущей культуры ОС загружает соответствующие ресурсы, а если они недоступны — загружаются ресурсы для культуры по умолчанию.

Примечание Назначать культуру пользовательскому интерфейсу следует до загрузки формы, отображающей локализованные ресурсы. Чтобы сделать это программно, надо установить культуру в конструкторе главной формы или методе *Main* приложения по крайней мере до отображения ключевых элементов пользовательского интерфейса.

Создание локализованных форм

.NET Framework позволяет создавать локализованные формы с затратой минимальных усилий. Все формы поддерживают свойство *Localizable*, которое определяет, является ли форма локализуемой. Установив это свойство в *true*, вы сделаете форму локализуемой.

Если свойство *Localizable* установлено в *true,* Visual Studio .NET автоматически создает для нее соответствующие файлы ресурсов. Установив это свойство в (*Defa-*

ult), вы зададите культуру по умолчанию для отображения интерфейса формы и ее элементов управления. Чтобы создать локализованную версию формы, назначьте свойству *Language* любое значение, отличное от (*Default*),— Visual Studio .NET автоматически создаст новый файл ресурсов для заданного языка и сохранит и, нем все значения, заданные для пользовательского интерфейса.

Примечание Обычно локализуют строковые свойства элементов пользовательского интерфейса, но в принципе это можно сделать для любого свойства. Например, заставить кнопку изменять ее размер в зависимости от длины надписи на ней (для разных культур она отличается), а графическое окно — его вид в зависимости от текущей культуры.

Чтобы просмотреть файл ресурсов формы, щелкните в окне Solution Explorer кнопку Show All Files и раскройте узел рядом с формой. Под формой отобразится список файлов ресурсов для всех языков, для которых имеются локализованные версии формы. На рис. 8-6 показан пример формы с четырьмя файлами ресурсов: для языка по умолчанию, английского, французского и немецкого.

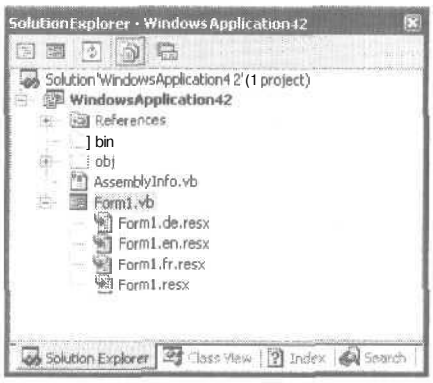


Рис. 8-6. **Файлы локализованных ресурсов**

▶ Создание локализованной формы

1. Установите свойство *Localizable* формы в *true*.
2. Создайте пользовательский интерфейс формы и переведите строковые и другие свойства его элементов на язык, для которого предоставляется локализация.
3. Создайте ресурсы интерфейса для культуры по умолчанию — они применяются, если культура не указана.
4. Задайте для свойства *Language* формы значение, определяющее соответствующую культуру.
5. Добавьте к вашей форме локализованное содержимое интерфейса.
6. Повторяйте пункты 4 и 5 для каждого языка, для которого планируется предоставить локализацию.
7. Выполните сборку приложения.

Если свойство *CurrentUICulture* определяет культуру, для которой есть локализация, приложение загрузит нужную версию формы, прочитав соответствующие файлы ресурсов. Если для указанной культуры файлов ресурсов не существует, интерфейсу назначается культура по умолчанию.

Проверка информации, вводимой на разных языках

Приложению может потребоваться механизм для проверки данных, вводимых на разных языках. Реализация в приложении, поддерживающем несколько языков, проверки вводимой с клавиатуры информации может оказаться нетривиальной задачей. Разработчик не всегда знает, какие символы используются в каждом языке, поддерживаемом приложением. А значит не сумеет корректно определить категорию символа, вводимого нажатием некоторых клавиш.

Методы, поддерживаемые структурой *Char,* позволяют проверять данные, введенные на любом языке, так же, как это делается для языков, использующих латинский алфавит. Методы *Char.IsDigit*, *Char.IsLetter* и ряд других возвращают корректные значения независимо от использованного при вводе набора символов, что обеспечивает корректность работы кода, основанного на применении этих методов, не требуя его адаптации для каждого из поддерживаемых языков.

Форматирование, зависимое от культуры

Для адаптации приложения к нуждам заказчика разработчику достаточно модифицировать некоторые члены класса *CultureInfo.* Предположим, что необходимо написать приложение для клиента, сотрудничающего с партнерами из Японии, однако денежные суммы в этом приложении должны исчисляться в американских долларах. Стало быть, приложение будет использовать для большинства элементов форматирование, принятое в японской культуре, но денежные суммы в нем всегда отображаются со знаком доллара.

Три члена класса *CultureInfo* определяют форматирование, применяемое при глобализации приложений (таблица 8-3). Модифицируя эти члены, можно создавать различные комбинации форматов, отсутствующие в стандартных культурах.

Таблица 8-3. Члены класса *CultureInfo,* определяющие форматирование

Член	Описание
DateTimeFormat	Содержит сведения о формате даты и времени, в том числе сокращенные названия дней недели, а также данные, определяющее отображение названий месяцев и времени в 12-часовом формате. Определяет форматы календаря, дней недели, дат и времени, применяемые при форматировании
NumberFormat	Содержит сведения о форматах чисел и денежных единиц, включая описания разделителей десятичных дробей и разрядов тысяч, применяемых для форматирования этих значений. Определяет символ, применяемый для отображения денежных единиц, и форматирование процентных значений
TextInfo	Содержит сведения о форматировании текста, включая кодовую страницу и символ, используемый в качестве разделителя элементов списков

Свойства этих членов разрешается модифицировать для создания пользовательских наборов форматов, отсутствующих в стандартных культурах. Вернемся к приведенному выше примеру: чтобы получить нестандартную культуру, состоящую в основном из параметров форматирования, принятых в Японии (за исключением денежных единиц, которые должны отображаться со знаком доллара), следует создать новый объект *CultureInfo* и соответствующим образом модифицировать его.

Следующий пример демонстрирует создание нового класса *CultureInfo* на основе параметров японской культуры и замену символа японской валюты знаком доллара:

Visual Basic .NET

```
' Этот пример предполагает наличие операторов Imports System.Globalization
' и Imports System.Threading
Dim modJPCulture As New CultureInfo("jp-JN")
modJPCulture.NumberFormat.CurrencySymbol = "$"
Thread.CurrentThread.CurrentCulture = modJPCulture
```

Visual C#

```
// Этот пример предполагает наличие операторов using System.Globalization
// и using System.Threading
CultureInfo modJPCulture = new CultureInfo("jp-JN");
modJPCulture.NumberFormat.CurrencySymbol - "$";
Thread.CurrentThread.CurrentCulture = modJPCulture;
```

Отображение текста справа налево

В некоторых языках пишут и читают справа налево, а не наоборот, как принято в большинстве языков, использующих латинский алфавит. Стандартные элементы управления Windows Forms поддерживают свойство *RightToLeft*, которое позволяет реализовать отображение текста пользовательского интерфейса приложения справа налево.

Свойство *RightToLeft* принимает одно из следующих трех значений — *Yes*, *No* и *Inherit*. По умолчанию свойство *RightToLeft* установлено в *Inherit*, при этом его реальное значение наследуется от родительского элемента управления.

Если установить свойство *RightToLeft* в *Yes*, вид элемента управления изменится, причем количество изменений зависит от типа элемента управления. Во-первых, по-другому выравнивается текст: обычно он выравнивается по левому краю, теперь — по правому. Во-вторых, заголовок элемента управления также выравнивается по правому краю. Вертикальная полоса прокрутки отображается слева, а при инициализации горизонтальных полос прокрутки ползунок позиционируется справа. Значение свойства *CheckAlign* для флажков изменится на противоположное, ярлычки вкладок располагаются другой стороны окна, выравнивание элементов с писков также изменится. Короче говоря с точки зрения форматирования все элементы управления станут зеркальными отражениями самих себя.

Однако, несмотря на все эти изменения, содержимое элементов управления попрежнему отображается слева направо. Представьте себе стандартный элемент управления *TextBox*, отображающее текст как обычно, то есть слева направо (рис. 8-7).

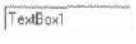


Рис. 8-7. Поле, отображающее текст слева направо

На следующем рисунке показано то же самое поле, но отформатированное подругому.

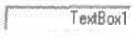


Рис. 8-8. Текстовое поле, отформатированное для отображения справа налево

Хотя выравнивание текста, отображаемого в поле, изменилось, сам текст по-прежнему отображается так же, как читается, — слева направо. Поэтому при локализации ресурсов для культур, в которых принято читать справа налево, придется отформатировать строки вручную.

Если установить свойство *RightToLeft* формы в *Yes,* то все размещенные на ней элементы управления, у которых *RightToLeft= Inherit,* будут выровнены по правому краю. Возможно, вам придется еще потрудиться и создать форму с зеркальным расположением элементов управления для культур, в которых принято читать и писать справа налево.

Для этого надо создать локализованную форму и вручную расположить на ней элементы управления — для каждого из них значения свойств, определяющие положение элемента управления на форме, будут записаны в файл ресурсов локализованной формы.

Преобразование кодировки символов

В .NET Framework для представления символов используется кодировка Unicode UTF-16, ставшая общепринятым стандартом. Она предоставляет уникальный код для каждого из более 65000 символов, существующих на данный момент, при этом остаются свободные коды еще для миллиона символов. Все это делает Unicode идеальной кодировкой для многоязычного содержимого.

До появления Unicode необходимость поддержки разных языков вынуждала разработчиков применять для внутреннего представления данных различные кодировки. В результате данные, отформатированные для различных культур, представлены в самых разных кодировках: однобайтовой для европейских языков, двухбайтовой для азиатских и двунаправленной для ближневосточных языков. Это затрудняет обмен данными между культурами и еще больше — разработку приложений с многоязычным интерфейсом.

.NET Framework позволяет преобразовывать данные из унаследованных форматов в формат Unicode, а при взаимодействии с унаследованными компонентами — из Unicode в унаследованные форматы.

Преобразования выполняет класс *Encoding* из пространства имен *System. Text.* Экземпляр этого класса нельзя создать напрямую, но можно получить посредством метода *Encoding. GetEncoding,* объявленного с модификатором доступа *Shared (static).* Он возвращает объект кодировки, представляющий кодовую страницу, например:

Visual Basic .NET

```
' Этот пример предполагает наличие оператора Imports System.Text
Din myEncoding As Encoding
' Кодовая страница 932 представляет японские иероглифы.
myEncoding = Encoding.GetEncoding(932)
```

Visual C#

```
// Этот пример предполагает наличие оператора using System.Text
Encoding  myEncoding;
// Кодовая страница 932 представляет японские иероглифы,
myEncoding = Encoding.GetEncoding(932);
```

Созданный экземпляр класса *Encoding* годится для преобразования символов из этой кодировки в Unicode и обратно. Чтобы выполнить преобразование существующих данных в Unicode, вызовите метод *Encoding.Convert*, передав ему объекты исходной и целевой кодировок, а также массив байт, представляющий данные в исходной кодировке. Этот метод возвращает массив байт, представляющих данные в целевой кодировке. Для преобразования в Unicode в качестве объекта целевой кодировки следует передать экземпляр класса *System.Text.UnicodeEncoding*.Следующий пример демонстрирует преобразование данных из кодовой страницы 932 в Unicode:

Visual Basic .NET

```
' Этот пример предполагает наличие оператора Imports System.Text.
' Исходные данные представлены массивом типа Byte.
' хранящимся в переменной myData.
Dim tgtData() As Byte
Dim srcEncoding As Encoding
Dim tgtEncoding As New UnicodeEncoding()
' Кодовая страница 932 представляет японские иероглифы.
srcEncoding = Encoding.GetEncoding(932)
' Теперь переменная tgtData содержит массив байтов, идентичный
' массиву myData, но закодированный в формате Unicode.
tgtData = Encoding.Convert(srcEncoding, tgtEncoding, myData)
```

Visual C#

```
// Этот пример предполагает наличие оператора using System.Text.
// Исходные данные представлены массивом типа Byte,
// хранящимся в переменной myData.
byte[] tgtData;
Encoding srcEncoding;
UnicodeEncoding tgtEncoding = new UnicodeEncoding();
// Кодовая страница 932 представляет японские иероглифы.
srcEncoding = Encoding.GetEncoding(932);
// Теперь переменная tgtData содержит массив байтов, идентичный
// массиву myData, но закодированный в формате Unicode.
tgtData = Encoding.Convert(srcEncoding, tgtEncoding, myData);
```

Чтобы преобразовать данные в унаследованном формате в массив байтом для дальнейшей обработки, вызовите метод *GetBytes*, передав ему объект исходной кодировки — в результате создается на основе символьного или строкового массива создается массив байтов:

Visual Basic .NET

```
' Этот пример предполагает наличие оператора Imports System.Text.
' Переменная myString содержит данные в унаследованном формате.
Dim myEncoding As Encoding
myEncoding = Encoding.GetEncoding(932)
Dim myBytes() As Byte
myBytes = myEncoding.GetBytes(myString)
```

Visual C#

```
// Этот пример предполагает наличие оператора using System.Text.
// Переменная myString содержит данные в унаследованном формате.
Encoding myEncoding;
myEncoding = Encoding.GetEncoding(932);
Byte[] myBytes;
myBytes = myEncoding.GetBytes(myString);
```

Метод *GetChars* позволяет преобразовать массив байтов в символьный массив, например:

Visual Basic .NET

```
' Этот пример предполагает наличие оператора Imports System.Text.
' Исходные данные находятся в массиве myBytes,
Dim myEncoding As New UnicodeEncoding()
Dim myChars() As Char
myChars = myEncoding.GetChars(myBytes)
```

Visual C#

```
// Этот пример предполагает наличие оператора using System.Text.
// Исходные данные находятся в массиве myBytes.
UnicodeEncoding myEncoding = new UnicodeEncoding();
char[] myChars;
myChars = myEncoding.GetChars(myBytes);
```

Резюме

- Под глобализацией понимают различное форматирование одних и тех же данных согласно правилам, принятым в различных культурах. Локализация — извлечение и отображение версии данных, соответствующих назначенной культуре.
- При глобализации приложений формат назначается данным динамически, а не «зашивается» в коде. При смене текущей культуры формат данных также изменяется.
- Локализацию реализуют посредством создания файлов с локализованными ресурсами для элементов пользовательского интерфейса. Если свойство *Localizable* формы установлено в *true,* то при изменении ее свойства *Language* автоматически создаются файлы ресурсов для данной формы.
- Экземпляры класса *CultureInfo* представляют отдельные культуры. Чтобы установить текущую культуру, запишите в свойство *System. Threading. Thread. CurrentThread. CurrentCulture* соответствующий экземпляр класса *CultureInfo.* Чтобы назначить культуру пользовательскому интерфейсу текущего потока, достаточно установить значение свойства *System. Threading. Thread. CurrentThread. CurrentUICulture.*
- Нестандартные культуры создают модификацией отдельных членов объекта *CultureInfo.*
- Метод *Encoding. Convert* позволяет преобразовать данные из некоторой кодировки в Unicode или любой другой формат. Чтобы генерировать массив байтов из массива строк или символов, вызовите метод *Encoding. GetBytes,* а метод *Encoding. GetChars* выполняет обратное преобразование.

Практикум 8. Создание локализованной формы, поддерживающей печать

Сейчас вы создадите простое приложение с многоязычным интерфейсом. Это приложение состоит из единственной формы, которая принимает заказы от пользователей и добавляет их в список. Вы дополните эту форму поддержкой глобализации, создадите локализованную версию формы для итальянских пользователей и в завершение организуете в созданном приложении поддержку печати. Решение задач этого практикума хранится в папке \Labs\Ch08\Solution на прилагаемом к книге компакт-диске.

Прежде **всего**

Изучение материалов этого практикума не требует предварительной подготовки.

Продолжительность практикума - около 60 минут.

Упражнение 1. Создание формы

Итак, ваше дело процветает, но вам не хватает программы для обработки поступающих заказов. Вы должны создать приложение, в котором пользователи смогут ввести количество и цену заказанных товаров. Итоги заказа приложение будет добавлять в список.

▶ Как создать форму

1. Создайте новый проект Windows Forms. Поместите на форму элементы управления, перечисленные в таблице 8-4, и установите для них свойства, указанные в этой таблице.

Таблица 8-4. Элементы управления и их свойства

Элемент управления	Свойство	Значение
Label1	*Name*	lblQuantity
	Text	Quantity
Label2	*Name*	lbl Price
	Text	Price
Label3	*Name*	lblTotal
	Text	Total
TextBox1	*Name*	txtQuantity
	Text	(пустая строка)
TextBox2	*Name*	txtPrice
	Text	(empty)
TextBox3	*Name*	txtTotal
	Text	(пустая строка)
	ReadOnly	True
Button1	*Name*	btnAdd
	Text	Add Order

Таблица 8-4. *(окончание)*

Элемент управления	Свойство	Значение
Button2	*Name*	btnClear
	Text	Clear Order
Button3	*Name*	btnClearAll
	Text	Clear All Orders
Button4	*Name*	btnPrint
	Text	Print
UstBoxl	*Name*	lstOrders

Форма с размещенными на ней элементами управления выглядит, как показано на рис. 8-9.

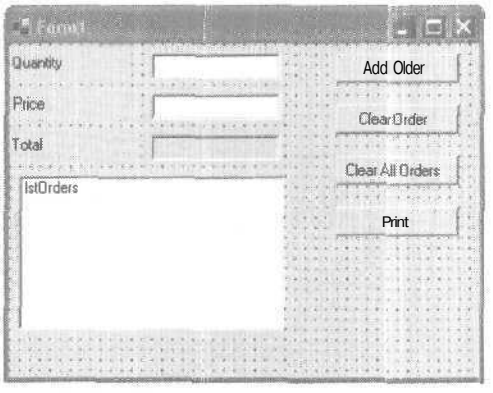


Рис. 8-9. Форма Lab Form1

2. Напишите обработчики для событий *KeyPress* элементов управления *txtPrice* и *txtQuantity,* проверяющие нажатие цифровых клавиш. Вот пример подобного обработчика для поля *txtPrice:*

Visual Basic .NET

```
Private Sub txtPrice_KeyPress(ByVal sender As Object, ByVal e As _
   System.Windows.Forms.KeyPressEventArgs) Handles txtPrice.KeyPress
   If Char.IsNumber(e.KeyChar) = False Then
      e.Handled = True
   End If
End Sub
```

Visual C#

```
private void txtPrice_KeyPress(object sender,
   System.Windows.Forms.KeyPressEventArgs e)
{
   if (char.IsNumber(e.KeyChar) == false)
      e.Handled = true;
}
```

3. Напишите обработчик события *Click* для кнопки *btnAdd*. Он должен умножать значения полей ***txtQuantity*** и *txtPrice* и отображать результат как денежную сумму в текстовом поле *txtTotal*. Кроме того, он должен добавлять строку, содержащую количество и цену товара, а также итоговую сумму, в список *lstOrder*. Вот примерный код этого обработчика:

Visual Basic .NET

```
Private Sub btnAdd_Click(ByVal sender As System.Object, ByVal e As _
   System.EventArgs) Handles btnAdd.Click
   Dim Total As Single
   Total = Single.Parse(txtQuantity.Text) * _
      Single.Parse(txtPrice.Text)
   txtTotal.Text = Format(Total, "Currency")
   Dim TotalString As String
   TotalString = txtQuantity.Text & " x " & txtPrice.Text & " = " _
      & txtTotal.Text()
   lstOrders.Items.Add(TotalString)
End Sub
```

Visual C#

```
private void btnAdd_Click(object sender, System.EventArgs e)
{
   float Total;
   Total = float.Parse(txtQuantity.Text) *
       float.Parse(txtPrice.Text);
   txtTotal.Text = Total.ToString("C");
   string TotalString;
   TotalString = txtQuantity.Text + " x " + txtPrice.Text + " = " +
      txtTotal.Text();
   lstOrders.Items.Add(TotalString);
}
```

4. Создайте обработчики для событий *btnClear* и *btnClearAll*. Первый сбрасывает значения только текстовых полей, а второй — сбрасывает значения и текстовых полей, и элементов списка.

Visual Basic .NET

```
Private Sub btnClear_Click(ByVal sender As System.Object, ByVal _
   e As System.EventArgs) Handles btnClear.Click
   txtQuantity.Text = "0"
   txtPrice.Text = "0"
   txtTotal.Text = Format(0, "Currency")
End Sub

Private Sub btnClearAll_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnClearAll.Click
      txtQuantity.Text = "0"
      txtPrice.Text = "0"
   txtTotal.Text = Format(0, "Currency")
```

```
    lstOrders.Items.Clear()
End Sub
```

Visual C#

```
private void btnClear_Click(object sender, System.EventArgs e)
{
    txtQuantity.Text = "0";
    txtPrice.Text = "0";
    txtTotal.Text = 0.ToString("C");
}
private void btnClearAll_Click(object sender, System.EventArgs e)
{
    txtQuantity.Text = "0";
    txtPrice.Text = "0";
    txtTotal.Text = 0.ToString("C");
    lstOrders.Items.Clear();
}
```

5. Сохраните и протестируйте проект. Форма будет вычислять стоимость заказа и добавлять результат в список, расположенный внизу формы.

Упражнение 2. Локализация формы

Дела идут хорошо, поэтом вы решили открыть филиал фирмы в другой стране (скажем, в Италии). Ваша задача — создать локализованную форму с интерфейсом на итальянском языке. Кроме того, необходимо реализовать возможность выбора языка перед началом работы с приложением.

▶ Создание локализованной версии формы

1. В окне дизайнера выберите форму Form1 и в окне Properties присвойте ее свойству *Localizable* значение *true*.

   Обратите внимание, что свойства *Language* по умолчанию установлено в *(Default)*. Текущие свойства пользовательского интерфейса станут параметрами, определяющими его облик при выборе языка по умолчанию.
2. В окне Properties поменяйте значение свойства *Language* с *(Default)* на *Italian (Italy)*.
3. В окне Properties установите для свойств, перечисленных в таблице 8-5, значения, указанные в этой таблице.

Таблица 8-5. Значения ресурсов для итальянской версии формы

Свойство	Значение
lblQuantity.Text	Quantita
lblPrice.Text	Prezzo
ibiTotal.Text	Totale
btnAdd.Text	Aggiungi ordine
btnClear.Text	Rimuovi ordine
btnClearAll.Text	Rimuovi tutti gli ordini
BtnPrint.Text	Stampa

Чтобы ввести символ «а», наберите 0224 на цифровой клавиатуре, удерживая нажатой клавишу Alt, либо воспользуйтесь стандартной программой Windows XP Character map (Таблица символов). Чтобы вызвать ее, щелкните Start\All Programs\Accessories\System Tools (Пуск\Все программы\Стандартные\Служебные).

Примечание Для корректного отображения содержимого на итальянском языке иногда приходится изменять размер некоторых элементов управления или формы.

4. В окне Properties установите свойство *Language* в *(Default)*. Заметьте, что надписям на элементах управления возвращается исходный вид.
5. В меню Project выберите команду Add Windows Form. Назовите новую форму **frmLanguage** и щелкните OK.
6. В окне дизайнера поместите на созданную форму поле со списком и кнопку, Установите для этих элементов управления свойства, как показано в таблице 8-6.

Таблица 8-6. Элементы управления формы Select Locale и их свойства

Элемент управления	Свойство	Значение
frmLanguage	Text	Choose your locale.
ComboBox1	Name	cmbLocale
	Text	Please select a locale.
	Items (add)	United States
	Items (add)	Great Britain
	Items (add)	Italy

7. В редакторе кода добавьте к форме *frmLanguage* обработчик события *SelectedIndexChanged* для элемента управления *cmbLocale*. Он должен установить назначать приложению и интерфейсу выбранную культуру и закрыть форму выбора языка. Вот примерный код этого обработчика:

Visual Basic .NET

```
Private Sub cmbLocale_SelectedIndexChanged(ByVal sender As _
   System.Object, ByVal e As System.EventArgs) Handles _
   cmbLocale.SelectedIndexChanged
   Dim aString As String
   aString = CType(cmbLocale.SelectedItem, String)
   Select Case aString
      Case "United States"
      ' Американский вариант английского является языком по умолчанию,
      ' поэтому дополнительных действий не требуется.
         Me.Close()
      Case "Great Britain"
         System.Threading.Thread.CurrentThread.CurrentCulture = _
            New System.Globalization.CultureInfo("en-GB")
         System.Threading.Thread.CurrentThread.CurrentUICulture _
            = New System.Globalization.CultureInfo("en-GB")
         Me.CloseO
      Case "Italy"
```

```
            System.Threading.Thread.CurrentThread.CurrentCulture = _
               New System.Globalization.CultureInfo("it-IT")
            System.Threading.Thread.CurrentThread.CurrentUICulture _
               = New System.Globalization.CultureInfo("it-IT")
            Me.Close()
      End Select
   End Sub
```

Visual C#

```
private void cmbLocale_SelectedIndexChanged(object sender,
   System.EventArgs e)
{
   string aString;
   aString = (string)cmbLocale.SelectedItem;
   switch (aString)
   {
      case "United States":
      // Американский вариант английского является языком по умолчанию,
      // поэтому дополнительных действий не требуется.
         this.Close();
         break;
      case "Great Britain";
         System.Threading.Thread.CurrentThread.CurrentCulture = new
            System.Globalization.CultureInfo("en-GB");
         System.Threading.Thread.CurrentThread.CurrentUICulture =
            new System.Globalization.CultureInfo("en-GB");
         this.CloseO;
         break;
      case "Italy":
         System.Threading.Thread.CurrentThread.CurrentCulture = new
            System.Globalization.CultureInfo("it-IT");
         System.Threading.Thread.CurrentThread.CurrentUICulture =
            new System.Globalization.CultureInfo("it-IT");
         this.CloseO;
         break;
   }
}
```

8. **Откройте форму Form1 в редакторе кода и найдите ее конструктор. Сразу после вызова конструктора базового класса [после строки MyBase.New() — в Visual Basic .NET] или в начале конструктора (Visual C#) добавьте следующий код:**

Visual Basic .NET

```
Dim aForm As New frmLanguage()
aForm.ShowDialog()
```

Visual C#

```
frmLanguage aForm = new frmLanguage();
aForm.ShowDialog();
```

Примечание Чтобы открыть конструктор формы в редакторе кода Visual Basic, следует развернуть область с пометкой «Windows Form Designer generated code».

9. Сохраните приложение и протестируйте его. В первой форме выберите регион «United States» и протестируйте приложение с этой версией интерфейса. Заметьте, что денежные суммы форматируются с использованием знака доллара ($), согласно формату, принятому в США. Закройте приложение и запустите его снова, На этот раз выберите «Great Britain». Теперь денежные суммы отображаются с использованием знака «фунт стерлингов» (£). Закройте приложение и запустите его еще раз, выбрав регион «Italy». Обратите внимание, что загружается локализованная версия формы с соответствующим заголовком. При этом денежные суммы отображаются с использованием знака евро согласно правилам, принятым в Европе: запятая является разделителем десятичных дробей, а точка отделяет разряды тысяч.

Упражнение 3. Создание поддержки печати

Сейчас вы дополните созданное приложение поддержкой печати. Для этого вы добавите компонент *PrintDocument* и напишете обработчик события *PrintPage*, отправляющий текст на принтер и поддерживающий печать многостраничных документов.

► Организация поддержки печати в приложении

1. Откройте форму Form1 в окне дизайнера и перетащите на нее объект *PrintDocument* с панели Toolbox — в области компонентов появится экземпляр одноименного класса.
2. Дважды щелкните компонент *Print Document 1 (print Document I)*, чтобы открыть обработчик по умолчанию для события *PrintPage*, и добавьте к нему следующий код, отправляющий содержимое списка *lstOrders* на принтер:

Visual Basic .NET

```
Dim LeftMargin As Single = e.MarginBounds.Left
Dim TopMargin As Single = e.MarginBounds.Top
Dim MyLines As Single = 0
Dim YPosition As Single = 0
Dim Counter As Integer = 0
Dim CurrentLine As String
Dim myFont As New Font("Times New Roman", 16, FontStyle.Regular, _
   GraphicsUnit.Pixel)
' Подсчитать число строк на странице.
MyLines = e.MarginBounds.Height / myFont.GetHeight(e.Graphics)
' Напечатать все строки файла, останавливаясь в конце каждой страницы.
While Counter < MyLines And ItemCounter <= lstOrders.Items.Count
   CurrentLine = CType(lstOrders.Items(ItemCounter), String)
```

```
    YPosition - TopMargin + Counter * myFont.GetHeight(e.Graphics)
    e.Graphics.DrawString(CurrentLine, myFont, Brushes.Black, _
       LeftMargin, YPosition, New StringFormat())
    Counter += 1
    ItemCounter += 1
End While
' Если остались ненапечатанные строки, напечатать следующую страницу.
If Not (ItemCounter - lstOrders.Items.Count - 1) Then
    e.HasMorePages = True
Else
    e.HasMorePages - False
End If
```

Visual C#

```
float LeftMargin = e.MarginBounds.Left;
float TopMargin = e.MarginBounds.Top;
float MyLines - 0;
float YPosition = 0;
int Counter = 0;
string CurrentLine;
Font myFont = new Font("Times New Roman", 16, FontStyle.Regular,
    GraphicsUnit.Pixel);
// Подсчитать число строк на странице.
MyLines = e.MarginBounds.Height / myFont.GetHeight(e.Graphics);
// Напечатать все строки файла, останавливаясь в конце каждой страницы.
while (Counter < MyLines && ItemCounter <=
    lstOrders.Items.Count - 1)
{
    CurrentLine = (string)lstOrders.Items[ItemCounter];
    YPosition = TopMargin + Counter * myFont.GetHeight(e.Graphics);
    e.Graphics.DrawString(CurrentLine, myFont, Brushes.Black, _
       LeftMargin, YPosition, new StringFormat());
    Counter += 1;
    ItemCounter += 1;
}
// Если остались ненапечатанные строки, напечатать следующую страницу.
if (!(ItemCounter == lstOrders.Items.Count - 1))
    e.HasMorePages - true;
else
    e.HasMorePages - false;
```

3. Сразу после объявления метода добавьте переменную *ItemCounter* — она будет управлять печатью элементов списка:

Visual Basic .NET

```
Dim ItemCounter As Integer
```

Visual C#

```
int ItemCounter;
```

4. В окне дизайнера формы Form1 дважды щелкните кнопку *btnPrint* и добавьте к открывшемуся обработчику вызов метода *PrintDocument.Print,* отправляющий содержимое списка на принтер. Установите переменную ItemCounter в ноль, чтобы при инициации печати список печатался полностью;

Visual Basic .NET

```
ItemCounter = 0
printDocument1.Print()
```

Visual C#

```
ItemCounter = 0;
printDocument1.Print();
```

5. Сохраните и протестируйте проект.

Закрепление материала

Приведенные ниже вопросы помогут вам лучше усвоить основные темы данной главы. Если вы не сумеете ответить на вопрос, повторите материал соответствующего занятия. Ответы для самопроверки — в приложении «Вопросы и ответы» в конце книги.

1. Расскажите, как вывести на печать документ при помощи компонента *Print Document,* выдержав межстрочные интервалы и, если это многостраничный документ, напечатать все его страницы?

2. Как с помощью методов *Begin* и *End* асинхронно вызвать метод Web-сервиса?

3. Перечислите и кратко поясните требования, необходимые для реализации специальных возможностей доступа, которые предъявляются к программам—кандидатам на получение логотипа «Certified for Windows».

4. Расскажите, как при помощи компонента *HelpProvider* снабдить элементы пользовательского интерфейса справочными сообщениями.

5. Как создать локализованную версию формы?

6. Как преобразовать данные из унаследованной кодировки в формат Unicode?

7. Чем отличаются глобализация и локализация?

ГЛАВА 9

Конфигурирование и защита сборок

Занятие 1. Сборки и ресурсы 394

Занятие 2. Конфигурирование и оптимизация приложений 405

Занятие 3. Защита приложений 413

В этой главе

Эта глава посвящена созданию сборок, их конфигурированию и защите от модификации злоумышленниками. Вы узнаете, как создавать закрытые, разделяемые сборки и сборки, предназначенные только для хранения ресурсов, а также о способах настройки и повышения производительности приложений с помощью конфигурационных файлов. В завершение речь пойдет о защите приложений и кода от несанкционированного доступа.

Прежде всего

Для изучения этой главы необходимо освоить материалы глав 1–8.

Занятие 1. Сборки и ресурсы

Сборка — это основная единица развертывания приложений, использующих .NET Framework. Она представляет собой совокупность логически связанных типов и ресурсов, составляющих функциональную единицу. Сборки являются *самоописываемыми*, поскольку содержат всю информацию, необходимую общеязыковой исполняющей среде для интерпретации их содержимого и настройки окружения для исполнения кода сборок. На этом занятии рассказывается о создании сборок и файлов ресурсов, о динамическом извлечении ресурсов во время выполнения, а также о совместном использовании сборки несколькими приложениями.

Изучив материал этого занятия, вы сможете:

- ✓ перечислить компоненты сборки;
- ✓ задать идентификационные данные сборки;
- ✓ создать библиотечную сборку;
- ✓ программно извлекать ресурсы;
- ✓ создавать сборки, содержащие только ресурсы;
- ✓ назначить сборке строгое имя;
- ✓ установить сборку в кэш глобальных сборок.

Продолжительность занятия - около 45 минут.

Сборки

Сборки представляют собой основные строительные блоки .NET-приложений. Они содержат типы и ресурсы, составляющие приложение, и описывают их для *общеязыковой исполняющей среды* (Common Language Runtime, CLR). Сборки обеспечивают многократное использование кода, управление версиями и защиту приложений, а также облегчают их развертывание.

Проще говоря, сборка — это проект, скомпилированный в EXE- или DLL-файл, Хотя внешне эти файлы не отличаются от тех, что создавались прежними версиями средств разработки, внутренняя структура этих файлов совершенно иная. Сборка состоит из четырех компонентов:

- **декларации сборки (ее метаданных)** — сведений о сборке, необходимых CLR;
- **метаданных типов** — сведений о типах, хранящихся в данной сборке;
- **кода** на промежуточном языке;
- **ресурсов** — неисполняемых данных, таких, как строки или изображения, специфичных для культуры.

Декларация сборки содержит метаданные, описывающие сборку для CLR, которая использует их для принятия решений, касающихся исполнения данной сборки. Декларация включает:

- **идентификационные данные сборки** — ее имя, номер версии и ряд дополнительных сведений, таких, как поддерживаемая культура и цифровая подпись;
- **список типов и ресурсов** — в декларации перечислены все типы, которые CLR может вызвать из этой сборки, и указан способ получения доступа к ним;
- **файлы** — полный список составляющих сборку файлов с указанием зависимостей между ними;

- **разрешения** — все разрешения, требуемые данной сборкой. Если локальная политика безопасности не позволяет получить запрошенные разрешения, исполнение кода сборки становится невозможным.

Как правило, разработчику не требуется разбираться в структуре декларации сборки — она создается и используется CLR автоматически. Тем не менее ему придется явно определить метаданные, которые описывают идентифицирующие сборку сведения.

Эти данные содержатся в файле AssemblyInfo.vb (AssemblyInfo.cs), который входит в состав проекта. Чтобы определить идентификационные данные сборки, щелкните правой кнопкой значок Assemblylnfo и выберите в раскрывающемся списке команду View Code — откроется редактор кода с содержимым файла Assemblylnfo. Он включает ряд атрибутов, идентифицирующих сборку, которые по умолчанию содержат Null-значения. Вот выдержка из файла Assemblylnfo (заметьте, что набор атрибутов по умолчанию для Visual Basic .NET и Visual C# отличается):

Visual Basic .NET

```
<Assembly: AssemblyTitle("")>
<Assembly: AssemblyDescription("")>
<Assembly: AssemblyCompany("")>
<Assembly: AssemblyProduct("")>
<Assembly: AssemblyCopyright("")>
<Assembly: AssemblyTrademark("")>
<Assembly: CLSCompliant(True)>
```

Visual C#

```
[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

Чтобы определить идентификационные данные сборки, установите значения соответствующих атрибутов в файле Assemblylnfo. Например, так устанавливается атрибут *AssemblyTitle*:

Visual Basic .NET

```
<Assembly: AssemblyTitle("The Best Assembly Ever!")>
```

Visual C#

```
[assembly: AssemblyTitle("The Best Assembly Ever!")]
```

Создание сборок с библиотеками классов

Часто требуются сборки, содержащие библиотеки классов. Они представляют собой набор типов, которые можно вызывать из других сборок. Например, в подоб-

ной сборке может быть нестандартный элемент управления, предназначенный для использования в нескольких приложениях, или компонент, поддерживающий функции высшей математики. Такую сборку нельзя исполнять непосредственно, но можно вызывать из других исполняемых приложений.

Сборки, содержащие библиотеки классов или элементы управления, создают на основе соответствующих шаблонов Visual Studio .NET. Шаблон для библиотеки классов позволяет создать сборку, содержащую типы, предназначенные для использования в других приложениях, а шаблон библиотеки Windows-элементов управления применяют для создания сборок с нестандартными элементами управления.

▶ **Как создать сборку с библиотекой классов**

1. В меню File выберите New\Project — откроется окно New Project
2. В окне New Project выберите Visual Basic Projects или Visual C# Projects, затем щелкните шаблон Class Library или Windows Control Library.
3. Напишите код библиотеки классов или Windows-элементов управления.
4. Задайте идентификационные данные сборки в ее файле AssemblyInfo.
5. В меню Build выберите команду Build, чтобы скомпоновать сборку.

Visual Studio .NET позволяет создать окончательную (Release) либо отладочную (Debug) версию сборки: щелкните меню Build\Configuration Manager и укажите требуемую версию. О настройке параметров сборки подробно рассказывается в занятии 2.

Ресурсы и ресурсные сборки

Большинству готовых программ требуются ресурсы. Термином «ресурсы» в данном контексте обозначают неисполняемые данные, внедренные в приложение. Пример — различные наборы строк или изображений, отображаемых пользовательским интерфейсом в зависимости от назначенной ему культуры. Упаковав подобные данные в ресурсный файл, вы получите возможность заменять их без перекомпиляции приложения. В главе 8 рассказывается об автоматической генерации ресурсных файлов и сборок при создании локализованных форм. Сейчас же речь пойдет о том, как вручную создать ресурсный файл и встроить его в сборку. Вы также узнаете, как создать сборку, содержащую только ресурсы, и научитесь программно извлекать код и объекты из таких сборок.

Создание ресурсных файлов

.NET Framework включает приложение-пример ResEditor, которое позволяет создавать ресурсные файлы, содержащие текст и изображения. Это приложение не интегрировано с IDE Visual Studio .NET, поэтому его следует запускать отдельно. На самом деле оно поставляется в виде исходного текста, поэтому перед использованием его необходимо скомпилировать. Файлы с исходным текстом вы найдете в каталоге \FrameworkSDK\Samples\Tutorials\resourcesandlocalization\reseditor, который расположен в каталоге, куда установлена Visual Studio .NET. Скомпилировать ResEditor можно при помощи командного файла, записанного в тот же каталог, либо импортировав код в проект Visual Studio и скомпоновав его после соответствующей настройки.

Примечание Если ResEditor компилируется с помощью командного файла, последний необходимо запускать из командной строки Visual Studio .NET — это гарантирует, что переменные окружения укажут путь к компилятору C#. Чтобы вызвать окно командной строки Visual Studio .NET в Microsoft Windows XP, щелкните Пуск\Программы\Visual Studio .NET\Visual Studio .NET Tools.

Инструмент ResEditor (его интерфейс показан на рис. 9-1) генерирует файлы с расширением .resources или .resx, содержащие строки и изображения.

ResEditor позволяет назначать типы и имена ресурсам, которые вы собираетесь включить в файл. Создав запись ресурса, можно добавить к ней собственно ресурс в виде строковых значений или изображений. Завершив формирование набора ресурсов, сохраните файл в двоичном формате (в этом случае ему назначается расширение .resources) или как XML-файл (тогда это будет .resx-файл).

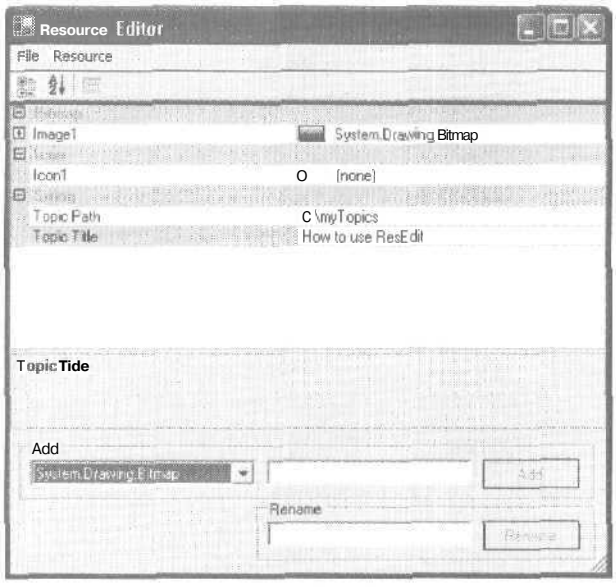


Рис. 9-1. Пользовательский интерфейс приложения ResEditor

▶ Создание файла ресурсов с помощью ResEditor

1. Скомпилируйте исходный текст ResEditor, если это еще не сделано. Файлы с исходным текстом хранятся в каталоге FrameworkSDK\Samples\Tutorials\resourcesandlocalization\reseditor, который находится в каталоге, куда установлена Visual Studio .NET.
2. Запустите ResEditor. Чтобы отредактировать существующий .resources- или .resx-файл, выберите в меню File команду Open и найдите нужный файл.
3. В раскрывающемся списке Add выберите тип добавляемого ресурса. Введите в поле Add имя нового ресурса и щелкните кнопку Add, чтобы добавить ресурс в файл.
4. В главном окне щелкните поле рядом с именем ресурса и задайте его значение. Если это строковый ресурс, введите соответствующую строку, а если это изображение, укажите файл изображения.

5. Чтобы переименовать ресурс, выделите его щелчком мыши, введите в поле новое имя и щелкните кнопку Rename, чтобы назначить ресурсу новое имя. Чтобы удалить ресурс, выделите его и выберите команду Delete из меню Resources.
6. Завершив редактирование файла ресурсов, шелкните команду File\Save As, чтобы сохранить его в формате .resources или .resx.

Созданный .resources- или .resx-файлы можно добавить к проекту, выбрав в меню Project команду Add Existing Item.

Внедрение ресурсов в сборки

Созданный файл ресурсов внедряют в сборку, чтобы упаковать ресурсы вместе с использующим их кодом — это улучшает переносимость приложения и уменьшает его зависимость от внешних файлов. Для этого достаточно всего добавить файл ресурсов к проекту приложения — добавленные ресурсы будут включены в сборку при компиляции проекта.

▶ **Как внедрить ресурсы в сборку**

1. Создайте файл ресурсов в формате .resources или .resx. Это удобно сделать с помощью ResEditor.
2. В меню Project выберите команду Add Existing Item — откроется одноименное окно.
3. Найдите файл ресурсов и щелкните кнопку Open — файл добавляется к проекту.
4. Из меню Build выберите команду Build, чтобы скомпилировать приложение, - создается сборка, в которую внедряются выбранные ресурсы.

Создание ресурсных сборок

Допустимо создавать сборки, содержащие только ресурсы. Это удобно, когда ресурсы нужно часто менять, но не хотелось бы перекомпилировать из-за этого все приложение.

Чтобы создать ресурсную сборку, добавьте файл ресурсов в пустой проект. При компоновке такого проекта создается сборка, содержащая исключительно ресурсы, которые можно применять в других сборках, ссылаясь на них в коде. Как получить доступ к ресурсам такой сборки — рассказано далее.

▶ **Как создать ресурсную сборку**

1. В меню File выберите команду New\Project — откроется окно New Project.
2. В этом окне выберите Visual Basic Projects или Visual C# Projects, затем Empty Project и щелкните OK — будет создан пустой проект.
3. В меню Project выберите команду Add Existing Item, чтобы добавить к новому проекту нужные файлы с ресурсами.
4. В окне Solution Explorer щелкните правой кнопкой этот проект и выберите из контекстного меню команду Properties — откроется окно свойств проекта.
5. В меню Output Type выберите для проекта выходной тип Class Library.
6. В меню Build выберите Build\<*имя_проекта*> — выполняется компиляция ресурсов в сборку.

Создание сопутствующих сборок

При создании приложений с многоязычным интерфейсом удобно использовать для каждой из поддерживаемых культур отдельный набор ресурсов. Это возможно благодаря *сопутствующим сборкам* (satellite assemblies), позволяющих автоматически загружать тот или иной набор ресурсов в зависимости от значения свойства *CurrentUICulture* потока. Из главы 8 вы узнали, как автоматически генерировать файл ресурсов для локализованной версии приложения, здесь же рассказывается о создании и подключении сопутствующих сборок для поддерживаемых культур.

В Visual Studio .NET создать сопутствующую сборку очень просто: достаточно добавить к приложению соответствующим образов названный файл с альтернативным набором ресурсов — остальное сделает Visual Studio .NET при компиляции проекта.

Имя файла ресурсов, встраиваемого в сопутствующую сборку, определяется культурой, для поддержки которой эти ресурсы предназначены. Оно составляется из имени файла с ресурсами для культуры по умолчанию, кода соответствующей культуры и расширения. Так, если файл ресурсов называется MyResources.resx, файл с ресурсами для поддержки немецкой культуры (для любых регионов) и именуется MyResources.de.resx, а для люксембургского варианта немецкого языка — MyResources.de-LU.resx.

Если добавить эти файлы к приложению, Visual Studio .NET автоматически скомпилирует их в сопутствующие сборки и поместит их соответствующие каталоги. Во время выполнения CLR автоматически загружает специфичные для культуры ресурсы, хранящиеся в этих файлах.

▶ Как создать сопутствующую сборку

1. Создайте файлы с альтернативными ресурсами для каждого из регионов, в которых предполагается использовать приложение.
2. Имя сопутствующей сборки должно включать имя файла ресурсов для культуры по умолчанию и код культуры, разделенные точкой.
3. В меню Project выберите команду Add Exiting Item, чтобы добавить к проекту созданные файлы.
4. Выберите из меню Build команду Build Solution, чтобы скомпоновать решение,
5. Файлы с ресурсами, специфичными для культуры, автоматически компилируются в сопутствующие сборки, которые записываются в соответствующие каталоги.

Извлечение ресурсов во время выполнения

Для извлечения внедренных ресурсов во время выполнения применяется класс *ResourceManager*, управляющий доступом и извлечением ресурсов из сборок. Каждый экземпляр класса *ResourceManager* связан со сборкой, содержащей ресурсы.

Конструктор класса *ResourceManager* требует два параметра: имя внедренного файла ресурсов (без расширения) и сборку, в которой он находится. Созданный экземпляр *Resource Manager* связывается с заданным файлом ресурсов, внедренным в сборку, имя этого файла (опять же без расширения) становится именем пространства имен, открывающего доступ к содержимому файла. Вот, например, как выглядит ссылка на ресурсы из файла myResources.de-DE.resx, расположенные в пространстве имен *Namespace1*, — *Namespace1.myResources*.

Второй параметр указывает сборку, в которой расположен данный файл ресурсов. Если это та же сборка, в которой находится объект, создающий экземпляр *ResourceManager,* можно получить ссылку на нее через тип этого объекта, например:

Visual Basic .NET

```
' Создание экземпляра ResourceManager для доступа к ресурсам из
' файла myResources.resx, расположенным в пространстве имен myNarnespace
' и внедренным в ту же сборку, что и текущий объект.
' Пример предполагает наличие оператора Imports System.Resources.
Dim myManager As New ResourceManager("myNamespace.myResources", _
   Me.GetType.Assembly)
```

Visual C#

```
// Создание экземпляра ResourceManager для доступа к ресурсам из
// файла myResources.resx, расположенным в пространстве имен myNarnespace
// и внедренным в ту же сборку, что и текущий объект.
// Пример предполагает наличие оператора, импортирующего System.Resources.
   ResourceManager myManager = new ResourceManager
      ("myNamespace.myResources", this.GetType().Assembly);
```

Если требуемые ресурсы находятся в другой сборке (например, в отдельной ресурсной сборке), перед извлечением ресурсов эту сборку необходимо загрузить. Сборки загружают при помощи экземпляра класса *Assembly* из пространства имен *System. Reflection*:

Visual Basic. NET

```
' Пример предполагает наличие оператора Imports System.Resources
Dim myResources As System.Reflection.Assembly
' Метод Assembly.Load принимает в качестве параметра
' имя загружаемой сборки.
myResources = System.Reflection.Assembly.Load("ResourceAssembly")
Dim myManager As New ResourceManager("ResourceAssembly.Resources", _
   myResources)
```

Visual C#

```
// Пример предполагает наличие оператора, импортирующего System.Resources
System.Reflection.Assembly myResojrces;
// Метод Assembly.Load принимает в качестве параметра
// имя загружаемой сборки.
myResources - System.Reflection.Assembly.Load("ResourceAssembly");
ResourceManager myManager = new
   ResourceManager("ResourceAssembly.Resources",
   myResources):
```

Примечание Прежде чем загружать сборку, необходимо добавить к проекту ссылку на нее.

Созданный объект-диспетчер ресурсов применяют для извлечения из файла ресурсов строк и объектов, Для извлечения строк применяют метод *ResourceManager.GetString*, требующий имя извлекаемого ресурса,

Метод *Resource Manager. GelObject* позволяет извлекать изображения и другие объекты. Он возвращает объект с заданным именем. Извлеченный объект необходимо явно преобразовать в требуемый тип. Вот как извлечь изображение:

Visual Basic .NET

```
' Этот пример предполагает наличие экземпляра класса ResourceManager
' с именем myManager.
Dim myImage As System. Drawing. Image
myImage = CType(myManager.GetObject("ImageResource"), _
   System.Drawing.Image)
```

Visual C#

```
// Этот пример предполагает наличие экземпляра класса ResourceManager
// с именем myManager.
System.Drawing.Image myImage;
myImage - (System. Drawing. Image)myManager.GetObject("ImageResource");
```

Извлечение из сопутствующих сборок ресурсов, специфичных для культуры

При наличии сопутствующих сборок с ресурсами, поддерживающими отдельные культуры, диспетчер ресурсов автоматически загружает соответствующие ресурсы в зависимости от значения свойства *CurrentUICulture*.Предположим, существует файл ресурсов myResources.resx и файл специфичных для культуры ресурсов myResources.de.resx. В этом случае диспетчер ресурсов загрузит ресурсы из файла myResources.de.resx, если *CurrentUICulture=de*.а при других значениях *CurrentUICulture* — ресурсы из файла myResources.resx.

▶ **Как извлечь внедренные ресурсы**

1. Создайте экземпляр класса *ResourceManager*, указав в качестве параметров пространство имен, имя внедренного файла с ресурсами (без расширения) и сборку, в которой находится этот файл.
2. Воспользуйтесь методом *Resource Manager. GetString*для извлечения строковых ресурсов и методом *ResourceManager.GetObject*для извлечения объектов. Помните: извлекаемые объекты необходимо явно преобразовывать в требуемый тип.

Разделяемыесборки

Сборки могут быть *закрытыми* (private) или *разделяемыми* (shared). Закрытая сборка используется только одним приложением, а разделяемая — доступна нескольким приложениями.

Примечание В данном контексте семантика терминов «закрытый» и «разделяемый» («открытый») отличается от таковой модификаторов доступа *Private* (*private*) и *Shared*, применяемых для объявления типов и членов. Здесь эти термины указывают, доступна ли некоторая сборка одному или нескольким приложениям.

Закрытые и разделяемые сборки

Большинство созданных вами сборок скорее всего будут закрытыми. Их проще создавать, к тому же по умолчанию создается именно этот тип сборок. Закрытая сборка доступна только одному приложению и является его компонентом, размещается вместе с приложением и доступна только из него. В связи с этим закрытые сборки не поддерживают управление версиями и расширенные идентификационные данные. Изучая материалы этого курса, мы до сих пор создавали только закрытые сборки.

Изучая элементы управления в главе 7, мы создавали сборки с нестандартными элементами управления, доступные из любых проектов, однако эти сборки являются закрытыми. Нет ли тут несоответствия?

Дело в том, что при добавлении в проект ссылки на закрытую сборку, Visual Studio .NET копирует DLL этой сборки в каталог проекта. Так, несколько проектов могут ссылаться на одну и туже DLL и вызывать содержащиеся в ней типы, но при этом у каждого такого проекта имеется своя копия этой DLL.

В отличие от закрытых, разделяемые сборки существуют на компьютере в единственном экземпляре, который доступен нескольким приложениям. Чтобы сделать сборку разделяемой, необходимо установить ее в *кэш глобальных сборок* (Global Assembly Cache, GAC). Хранение сборок в GAC имеет ряд преимуществ:

- **централизованное хранение сборок** — если некоторая сборка требуется нескольким приложениям, достаточно сделать ее разделяемой, чтобы решить проблему;
- **безопасность** — **GAC** находится в каталоге C:\W1NNT (в Microsoft Windows 2000) или WINDOWS (в Microsoft Windows XP), который по умолчанию защищен самыми строгими мерами безопасности;
- **возможность использовать несколько версий сборки одновременно** — в GAC можно устанавливать несколько версий одной и той же сборки. Каждое приложение определяет и загружает нужную ему версию.

Как говорилось ранее, чаще всего создают закрытые сборки. Разделяемые сборки требуются, если на то есть веские причины. Чтобы создать разделяемую сборку и установить ее в GAC, необходимо назначить сборке строгое имя.

Строгое имя

Строгое имя (strong name) гарантирует подлинность сборки. Оно включает идентификационные данные сборки, такие, как ее имя, номер версии, сведения о культуре (если таковая назначена) и открытый ключ из криптографической пары ключей. Все эти сведения шифруются закрытым ключом из данной пары. Полученные зашифрованные данные можно расшифровать соответствующим открытым ключом. Поскольку закрытый ключ есть лишь у автора сборки, никому кроме него не удастся создать копию строгого имени, что гарантирует подлинность сборки.

Чтобы назначить сборке строгое имя, требуется пара ключей — открытый и закрытый. При необходимости для их генерации применяют утилиту strong name (sn.exe).

▶ **Генерация пары криптографических ключей**

1. Вызовите командную строку Visual Studio .NET. В Windows XP для этого достаточно щелкнуть Пуск\Программы\Microsoft Visual Studio .NET\Visual Studio .NET Tools.

2. Наберите в командной строке sn.exe с ключом -k , чтобы указать файл, кула следует записать генерированные ей ключи. Обычно этот файл имеет расширение ,snk, например:

```
sn -k myKey.snk
```

В результате исполнения этой команды генерируется файл с парой ключей, который необходим для создания строгого имени.

▶ **Как подписать сборку с помощью строгого имени**

1. В Solution Explorer откройте файл AssemblyInfo.
2. Убедитесь, что для сборки задан номер версии при помощи атрибута *AssemblyVersion*, например;

Visual Basic .NET

```
<Assembly: AssemblyVersion("1.0.1.1")>
```

Visual C#

```
[assembly: AssemblyVersion("1.0.1.1")]
```

По умолчанию атрибут *Assembly Version* установлен в «1.0.*». Звездочка показывает, что CLR автоматически генерирует значения по умолчанию для последних двух цифр номера версии.

3. При помощи атрибута *AssemblyKeyFileAttribute*укажите путь (абсолютный или относительный) к файлу с ключами, например, так:

Visual Basic .NET

```
<Assembly: AssemblyKeyFile(". ,\. .\myKey.snk")>
```

Visual C#

```
[assembly: AssemblyKeyFile("..\\..\\myKey.snk")]
```

Этот атрибут автоматически добавляется в файл Assemblylnfo при использовании Visual C#; в Visual Basic .NET его придется добавить самостоятельно.

4. Скомпилируйте сборку — она будет подписана при помощи сгенерированного строгого имени.

Установка сборок в GAC

Процедура установки в GAC сборки со строгим именем проста и выполняется с помощью утилиты gacutil.exe.

▶ **Как установить сборку в GAC**

1. Подпишите сборку при помощи строгого имени (об этом рассказано в предыдущем разделе).
2. Вызовите командную строку Visual Studio .NET, щелкнув Пуск\Программы\ Microsoft Visual Studio .NET\Visual Studio .NET Tools.
3. Вызовите утилиту gacutil.exe с параметром /i, указав сборку для установки в GAC. Вот как установить в GAC сборку myAssembly.dll;

```
gacutil /i myAssembly.dll
```

Резюме

- Сборки — это строительные блоки приложений. Сборка состоит из метаданных сборки (декларации), метаданных типов, файлов кода и ресурсов. Проще говоря, сборка — это результат компиляции любых проектов библиотек и исполняемых файлов.
- Сборки с библиотеками классов в Visual Studio .NET создают на основе шаблона библиотеки классов или библиотеки Windows-элементов управления.
- Ресурсы — это неисполняемые данные, логически связанные с приложением. Ресурсы добавляются к приложению в виде .resources- или .resx-файлов. При компиляции файлы ресурсов внедряются в сборку.
- Чтобы создать сборку с ресурсами, просто сгенерируйте новый проект и добавьте к нему нужные файлы с ресурсами. DLL, созданная в результате компиляции такого проекта, содержит ресурсы, доступные из других сборок.
- Для поддержки различных культур создают файлы с альтернативными версиями ресурсов. Имена этих файлов должны содержать коды соответствующих культур. При компиляции проекта из них автоматически создаются сопутствующие сборки.
- Для извлечения ресурсов из сборки во время выполнения применяется класс *Resource Manager*. Каждый экземпляр этого класса связан с файлом ресурсов, внедренным в некоторую сборку. Файл и сборку задают как параметры конструктора класса *Resource Manager*. Если для культуры, заданной свойством *CurrentUICulture*, имеется сопутствующая сборка, объект-диспетчер ресурсов автоматически извлекает из нее нужные ресурсы.
- Большинство сборок являются закрытыми. Такая сборка доступна только приложению, с которым она связана. При добавлении в проект ссылки на закрытую сборку последняя копируется в каталог этого проекта.
- Разделяемые сборки доступны сразу нескольким приложениям. Разделяемую сборку следует установить в кэш глобальных сборок (GAC). Для этого сборке необходимо сначала назначить строгое имя, после чего ее можно поместить в GAC средствами утилиты gacutil.exe.

Занятие 2. Конфигурирование и оптимизация приложений

В .NET Framework имеется ряд инструментов для конфигурирования и настройки приложений после развертывания. Конфигурационные файлы позволяют указывать местонахождение и версию зависимых сборок, а также установить другие свойства приложения. Специальная утилита для мониторинга производительности позволяет выявлять «узкие места» приложений, позволяя устранить их в следующих версиях.

Сейчас вы научитесь создавать конфигурационные файлы и применять их для установки динамических свойств приложений, а также выявлять «узкие места» в коде приложения с помощью монитора производительности.

Изучив материал этого занятия, вы сможете:

- ✓ создать конфигурационный файл;
- ✓ описать базовую схему конфигурационного файла;
- ✓ объяснить, как установить динамические свойства приложения;
- ✓ сконфигурировать динамические свойства приложения;
- ✓ выявлять «узкие места» в приложении с помощью утилиты perfmon.exe.

Продолжительность занятия - около 30 минут.

Приложение можно снабдить конфигурационным файлом, что позволит изменять свойства приложения после развертывания, не прибегая к его перекомпиляции. Конфигурационный файл — это файл в формате XML, содержащий информацию о параметрах приложения. О настройке трассировочных переключателей с помощью конфигурационного файла говорилось в главе 5, а здесь мы обсудим настройку других параметров работы приложения.

Создание конфигурационного файла

Конфигурационный файл — это просто XML-файл с соответствующим именем, содержащий набор необходимых тегов. Имя конфигурационного файла приложения должно иметь следующий вид: <*name*>.<*extension*>.config, где <*name*> — имя, а <*extension*>— расширение исполняемого файла приложения (например, .exe). Так, конфигурационный файл приложения myApplication.exe назван myApplication.exe.config. Конфигурационный файл должен располагаться в одном каталоге со сборкой приложения, для настройки которого он предназначен.

Содержимое конфигурационных файлов определяется общей для этих файлов схемой. Базовая структура конфигурационного файла выглядит так:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <!-здесь находится описание параметров приложения->
</configuration>
```

Обязателен только первый тэг, в котором указаны версия XML, используемая кодировка и элемент <configuration> верхнего уровня, остальные элементы не обязательны и добавляются или удаляются по мере необходимости.

Чтобы создать конфигурационный файл для сборки, написанной на Visual Basic .NET, выберите в меню File пункт Add New Item\Application Configuration File – к проекту будет добавлен новый конфигурационный файл, в который можно вручную добавить необходимые элементы. При компиляции приложения файлу назначается соответствующее имя.

При использовании Visual C# конфигурационный файл придется создавать и заполнять вручную при помощи текстового редактора (например, Блокнота). Готовый конфигурационный файл следует сохранить с именем app.config и добавить к проекту приложения.

Схема конфигурационного файла

Полностью обсудить схему конфигурационного файла в этом разделе на представляется возможным, но ее элементы верхнего уровня мы все же рассмотрим. Более подробно об этом рассказано в документации по Visual Studio .NET. Элементы верхнего уровня схемы конфигурационного файла описаны в таблице 9-1.

Таблица 9-1. Элементы верхнего уровня схемы конфигурационного файла

Элемент	Что содержит
<startup>	Единственный элемент — *<requiredRuntime>*, задающий требуемую версию CLR
<runtime>	Параметры, управляющие связыванием сборок и сбором мусора
<system.runtime.remoting>	Сведения о конфигурации каналов и удаленных объектах
<system.net>	Сведения для Интернет-приложений
<mscorlib> Элемент	*< cryptography Settings>*, определяющий, как приложение использует криптографические технологии
<configSections>	Нестандартные параметры
<system.diagnostics>	Конфигурационные данные для классов *Trace a Debug*

▶ **Создание конфигурационного файла для приложения на Visual Basic .NET**

1. В меню Project выберите пунктAdd New Item — откроется одноименное окно.
2. В окне Add New Stem выберите шаблон Application Configuration File — к проекту будет добавлен конфигурационный файл.
3. Внутри элемента <configuration> поместите элементы схемы для настройки необходимых параметров. Более подробно о доступных элементах схемы — в документации на Visual Studio .NET.
4. Сохраните созданный файл и скомпонуйте приложение.

▶ **Создание конфигурационного файла для приложения на Visual C#**

1. В меню Project выберите Add New Item.
2. В окне Add New Item выберите Text File — новый текстовый файл добавляется к проекту и открывается в текстовом редакторе.
3. **В окне** Solution Explorer щелкните правой кнопкой новый текстовый файл и выберите команду Rename, чтобы переименовать файл в **App.config.** В текстовом редакторе добавьте к файлу следующий XML-код:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

В окне Solution Explorer дважды щелкните файл App.config. Ответьте согласием на вопрос о закрытии файла — среда разработки переключится в редактор с XML-текстом файла App.config.

Динамические свойства

Динамические свойства приложения позволяют инициализировать его объекты. Так, некоторые свойства объектов приложения можно связать с элементами конфигурационного файла и динамически извлекать для них значения во время выполнения. При помощи динамических свойств также удобно связывать внешние ресурсы, изменяющиеся на протяжении жизненного цикла приложения, такие, как строки подключения к базе данных. Эти свойства позволяют изменять параметры приложения во время выполнения, не прибегая к его перекомпиляции и повторному развертыванию. В Visual Studio .NET динамические свойства можно задавать во время разработки через окно Properties, а код для извлечения значений динамических свойств во время выполнения вам придется написать самостоятельно.

Настройка динамических свойств посредством окна Properties

Чтобы сделать избранные свойства элементов пользовательского интерфейса динамическими, воспользуйтесь окном Properties. Для каждого элемента управления это окно отображает узел DynamicProperties (рис. 9-2), который позволяет сделать данное свойство динамическим. По умолчанию этот узел уже содержит свойства, которые скорее всего понадобится связать с внешними ресурсами. Чтобы вручную добавить свойства в этот список, щелкните кнопку рядом с (Advanced).

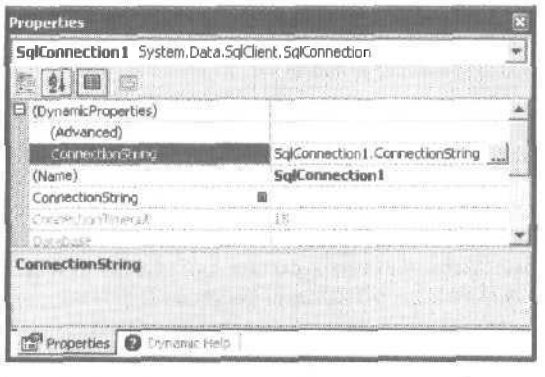


Рис. 9-2. Узел DynamicProperties, отображаемый в окне Properties

Чтобы заставить некоторое свойство динамически читать значение из конфигурационного файла, необходимо предоставить ключ. Он автоматически записывается в конфигурационный файл и идентифицирует нужное значение. Вот пример элемента, который Visual Studio .NET добавила в .config-файл, чтобы сделать свойство *Button1.Text* динамическим:

```
Odd key="Button1.Text" value="Button1" />
```

Приложение использует значение ключа (*Button1.Text)* для извлечения значения (*Button1*) свойства *Button1.Text* во время выполнения. Поскольку значения ключей должны быть понятны человеку, их лучше создавать по определенной схеме. В Visual Studio .NET по умолчанию действует схема следующего вида: <*имя_элемента_управления*>.<*имя_динамического_свойства*>.

Любые динамические свойства можно модифицировать после развертывания приложения, непосредственно редактируя конфигурационный файл. Например, чтобы изменить значение свойства *Button1.Text* с «Button1» на «myButton», достаточно найти и отредактировать соответствующий элемент .config-файла. При следующем запуске приложения из конфигурационного файла будет прочитано новое значение для данного свойства.

▶ **Определение динамических свойств через окно Properties**

1. В окне дизайнера выберите нужный элемент управления.
2. В окне Properties раскройте узел Dynamic Properties.
3. Если требуемое свойство уже есть в списке этого узла, щелкните кнопку рядом с его именем и введите ключ для этого свойства.
4. Если же это свойство еще не добавлено в список узла Dynamic Properties, щелкните кнопку рядом со строкой (Advanced) — откроется окно Dynamic Properties.
5. В окне Dynamic Properties щелкните свойство, которое вы хотите сделать динамическим, и выберите для него ключ из раскрывающегося списка, — значения выбранных свойств и указанные для них ключи записываются в конфигурационный файл.

Примечание В списке узла Dynamic Properties доступны не все свойства. Поскольку в конфигурационных файлах значения свойств хранятся в виде строк, динамическими можно сделать лишь свойства строкового или иного типа, который удастся явно преобразовать в строковый.

▶ **Как установить динамическое свойство приложения после его развертывания**

1. Откройте конфигурационный файл приложения в Блокноте или в другом текстовом редакторе.
2. Найдите элемент свойства, которое следует модифицировать.
3. Отредактируйте атрибут *Value* этого элемента, записав в него требуемое значение.

Примечание Значения свойств, которые по своей сущности не являются строковыми, например булевы свойства, следует задавать в виде строк, которые при синтаксическом разборе дают значения требуемого типа. Например, значения свойства типа *Boolean* надо задавать в виде строки *true* или *false*.

4. Сохраните и закройте конфигурационный файл. При следующем запуске приложение прочитает из него новое значение свойства.

Установка и извлечение динамических свойств вручную

Иногда требуется сделать динамическим свойство объекта, который не относится к элементам пользовательского интерфейса. Рассмотрим класс, экземпляр которого создается во время выполнения. При этом ряд свойств нового объекта требуется

инициализировать значениями, зависящими от внешних факторов. В данном случае необходимые значения можно задать через конфигурационный файл и динамически извлекать их во время выполнения при помощи класса ***AppSettingsReader***.

Класс *AppSettingsReader* определен в пространстве имен *System.Configuration* и применяется для извлечения из конфигурационного файла значений с заданным ключом. Главный метод этого класса — *GetValue*. Он принимает как параметры строку ключа и тип извлекаемого объекта. Хотя тип извлекаемого объекта определяется вторым параметром, этот метод возвращает значение типа *Object*, поэтому его необходимо явно преобразовать в требуемый тип. Вот пример, демонстрирующий установку свойства *Text* гипотетического объекта *Widget* с помощью класса *AppSettings Reader*.

Visual Basic .NET

```
' Создать объект AppSettingsReader.
Dim myReader As New System.Configuration.AppSettingsReader()
' Создать объект Widget.
Dim myWidget As New Widget()
' Извлечь значение динамического свойства с ключом DynamicWidget.Text
' и типом String и получить результат, преобразовав возвращенный Object
' в String при помощи функции CType.
myWidget.Text = CType(myReader.GetValue("DynamicWidget.Text", _
   GetType(System.String)), String)
```

Visual C#

```
// Создать объект AppSettingsReader.
System.Configuration.AppSettingsReader myReader = new
   System.Configuration.AppSettingsReader();
// Создать объект Widget.
Widget myWidget = new Widget();
// Извлечь значение динамического свойства с ключом DynamicWidget.Text
// и типом String и получить результат, преобразовав возвращенный Object
// в String при помощи функции CType.
myWidget.Text = myReader.GetValue("DynamicWidget.Text",
   typeof(System.String)).ToString();
```

При попытке чтения из конфигурационного файла значения с несуществующим ключом генерируется исключение *InvalidOperation Exception*.

Чтобы извлечь данные из конфигурационного файла во время выполнения, необходимо добавить к нему все необходимые элементы во время выполнения. Эти элементы заключаются в тэг <appSettings> и состоят из тэга <add> с парой «ключ — значение». Вот как добавить элемент значения в конфигурационный файл:

```
<appSettings>
   <!-- Сюда помещают параметры приложения, определяемые
        пользователем. -->
   <!-- Вот пример: <add key="ключ" value="значение_свойства"/> -->
   <add key="Widget.Visible" value="True" />
   <add key="Widget.Text" value="I love my Widget!" />
</appSettings>
```

▶ **Как вручную извлечь данные из конфигурационного файла**

1. Создайте экземпляр класса *System.ConfigurationAppSettingsReader*.
2. Вызовите метод *AppSettingsReader.GetValue*, чтобы извлечь значение с заданным ключом.
3. Преобразуйте объект, возвращаемый методом *AppSettingsReader.GetValue*, в требуемый тип.

▶ **Как вручную записать данные в конфигурационный файл**

1. Откройте конфигурационный файл приложения в Блокноте или другом текстовом редакторе.
2. Найдите тэг <appSettings>; если он отсутствует, определите его внутри элемента <configuration>.
3. Добавьте к <appSettings> элемент <add> с атрибутами ключа и значения, как показано ниже:

```
<add key="Widget1.Text" value="My Widget">
```

4. Сохраните и закройте конфигурационный файл.

Оптимизация приложений

Работа приложением не прекращается даже после его компоновки и развертывания. В ответ на меняющиеся требования бизнеса приложения приходится постоянно усовершенствовать с целью повышения их производительности и более рационального использования ресурсов. В этом разделе приводится ряд полезных правил оптимизации приложений.

Оптимизация на этапе разработки

Эффективный и качественный код является результатом тщательного планирования и следования известным правилам программирования. Придерживайтесь их, и вам удастся создавать рационально спроектированные приложения.

- **Избегайте позднего связывания.** То есть использования типа *Object (object)*. Лишние операции преобразования типов требуют много ресурсов и снижают производительность приложения. Тем. кто пишет на Visual Basic .NET, рекомендуется всегда применять параметр Option Strict On — он включает строгую проверку типов, уменьшая число неоправданных преобразований.
- **Старайтесь не использовать глобальные переменные,** по возможности заменяйте их локальными переменными или константами. Из-за особенностей выделения памяти под глобальные переменные получить к ним доступ всегда сложнее, чем к локальным. Глобальные переменные следует использовать лишь там, где без них не обойтись. Представление значений, часто встречающихся в коде, при помощи констант также повышает производительность.
- **Внимательно относитесь к циклам,** поскольку именно при их обработке приложения выполняют значительную часть операций. Циклы необходимо тщательно продумывать и стараться помешать в них как можно меньше действий.

Стратегия оптимизации

Типичный план оптимизации кода приложения таков:

1. измерьте показатели производительности;
2. выявите «узкие места», проанализировав полученные данные;

3. устраните их, изменив код приложения;
4. повторите пункты 1–3.

Обнаружив «узкие места», проанализируйте соответствующие фрагменты кода приложения для поиска возможных мер по улучшению производительности. Реализовав эти меры, снова оцените производительность приложения, чтобы найти новые «узкие места» и узнать, удалось ли повысить производительность. Устранив обнаруженные недостатки, продолжайте мониторинг — это позволит выявить другие, возможно менее серьезные недостатки, и более усовершенствовать приложение.

Измерение производительности

Утилита perfmon.exe в Windows 2000 и Windows XP применяется для мониторинга производительности. Она позволяет измерять широкий спектр параметров производительности, отображая собранные данные в виде диаграммы или записывая их в файл журнала.

Для наблюдения за производительностью приложений также используют оператор *Trace*. Генерированные операторами *Trace* данные отражают различные события в ходе исполнения приложения. Сопоставив трассировочные данные с результатами мониторинга производительности, зачастую удается обнаружить код, обусловивший появление «узкого места».

Оптимизация средствами компилятора

Оптимизировать приложения позволяют и некоторые функции компилятора. Естественно, это не освобождает от необходимости писать качественный код, но иногда оптимизирующие возможности компилятора оказываются весьма полезны разработчику.

Оптимизирующие функции компилятора включаются на странице свойств приложения (рис. 9-3). Чтобы открыть его, в окне Solution Explorer щелкните правой кнопкой проект и выберите из контекстного меню команду Properties. Если вы пишете на Visual Basic .NET, выберите в каталоге Configuration Properties папку Optimizations,

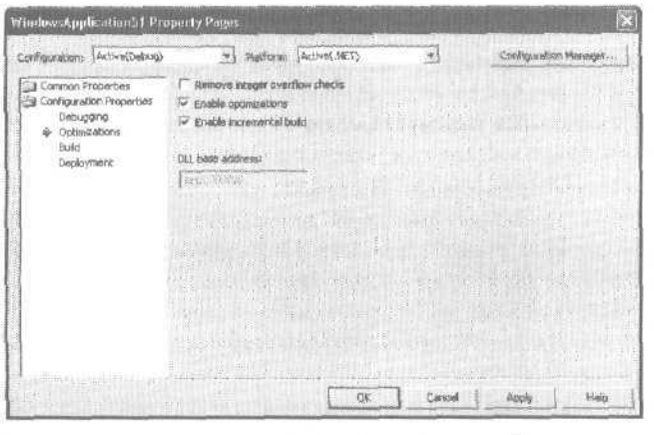
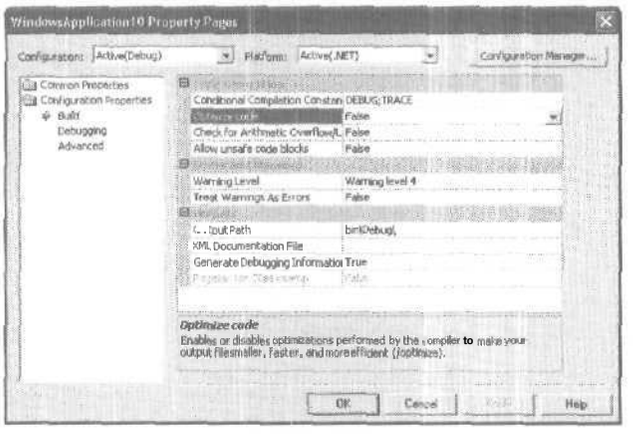


Рис. 9-3. Страница Optimizations Property Page (для приложения на Visual Basic .NET)

Если же вы используете Visual C#, выберите папку Build в каталоге Configuration Properties (рис. 9-4).

Чтобы включить оптимизирующие функции компилятора, пометьте флажок Enable Optimizations (Visual Basic .NET) или установите параметр Optimize Code в Тше (Visual C#) — тогда при компиляции код приложения будет оптимизироваться. При применении оптимизирующих функций компилятора приложение модифицируется на уровне IL-кода, что затрудняет его отладку, поэтому лучше включать оптимизацию при компиляции полностью отлаженного приложения в окончательную сборку.



Рис, 9-4. Страница Build Property Page (для приложения на Visual C#)

Резюме

- Прииспользовании конфигурационного файла можно изменить параметры конфигурации приложения без его перекомпиляции — достаточно отредактировать конфигурационный файл в текстовом редакторе, сохранить внесенные изменения и перезапустить приложение.
- Динамические свойства удобны для описания изменяющихся внешних ресурсов уже готового приложения. Свойство элемента пользовательского интерфейса делают динамическим в дизайнере (средствами окна Properties).
- Для извлечения данных из конфигурационного файла вручную применяют класс *AppSettingsReader*. Объект *AppSettingsReader* извлекает данные из содержащихся в файле ресурсов тэгов <add>, используя заданный ключ для поиска необходимого значения.
- Мониторинг производительности приложений осуществляют при помощи утилиты perfmon.exe и операторов *Trace*. Оптимизации приложений способствует рациональное программирование и последовательные измерения производительности с дальнейшим исправлением выявленных недостатков программы.
- Чтобы задействовать оптимизирующие функции компилятора, на странице свойств приложения установите флажок Enable Optimizations (Visual Basic .NET) или выберите параметр Optimize code (Visual C#). Оптимизирующие возможности компилятора следует применять только при создании окончательной версии сборки.

Занятие 3. Защита приложений

При разработке приложения важно не забыть о его защите. Атаки хакеров и злоумышленников ежегодно наносят огромный ущерб бизнесу. Разработчики должны создавать код, устойчивый к атакам злонамеренных лиц. Кроме того, безопасность приложений подразумевает защиту конфиденциальной информации, такой, как корпоративные базы данных и персональная информация, от несанкционированного доступа. .NET Framework предоставляет для этого широкий спектр инструментов. На этом занятии описаны императивная и декларативная защита, приемы защиты от несанкционированного доступа пользователей и программ, а также от действий неидентифицированного кода. Кроме того, вы научитесь реализовывать нестандартные меры безопасности.

Изучив материал этого занятия, вы сможете:

- ✓ рассказать, что такое разрешения и для чего они нужны;
- ✓ объяснить различия между декларативной и императивной защитой;
- ✓ конфигурировать защиту, основанную на ролях;
- ✓ конфигурировать защиту по правам доступа к коду,

Продолжительность занятия - около 40 минут.

Обеспечение безопасности приложения сводится к защите от несанкционированного доступа. Поддерживаемые .NET Framework средства обеспечения безопасности позволяют защищать код приложения от доступа пользователей, не обладающих соответствующими полномочиями, и запретить системе несанкционированное исполнение кода приложения. Общесистемную политику безопасности определяет системный администратор. Он разрешает или запрещает исполнение программ, устанавливает отношения доверия для сборок и определяет их характер. Установленную системным администратором политику безопасности невозможно изменить из кода приложения — она обеспечивает наивысший уровень зашиты на данном компьютере.

Разработчик использует доступные ему средства защиты для реализации дополнительных мер безопасности в рамках политики, заданной системным администратором. Для проверки наличия у пользователей и программ права на доступ к приложениям применяют модель системы безопасности, основанную на ролях. Проверка удостоверений безопасности бывает императивной либо декларативной. При использовании императивной защиты разрешения на исполнение кода запрашиваются во время выполнения, а в случае применения декларативной зашиты требуемые разрешения указывают в декларации сборки. Если запрашиваемые разрешения входят в противоречие с системной политикой безопасности, сборке будет отказано в предоставлении разрешения на исполнение. Объекты разрешений — это центральные объекты инфраструктуры защиты кода.

Примечание Если не указано обратное, во всех приведенных в этом занятии примерах кода подразумевается наличие оператора *Imports System.Security.Permissions* (Visual Basic .NET) или *using System.Security.Permissions*(Visual C#).

Разрешения

Разрешения (permissions) — основные объекты системы безопасности. Это программная сущность, представляющая пользователя, удостоверение безопасности или программный ресурс. Такие объекты применяются для исполнения самых разных действий, связанных с обеспечением безопасности, — проверки прав доступа и реализации политики безопасности.

Интерфейс IPermission

Во всех объектах разрешений реализован интерфейс *IPermission*. Вам, как разработчику, редко придется реализовывать этот интерфейс в своих объектах, не исключено, что он вам вовсе не понадобится. Тем не менее, изучив объявленные в *IPermission* методы (таблица 9-2), вы лучше поймете, как работают объекты, в которых реализован этот интерфейс.

Таблица 9-2. Методы интерфейса *IPermission*

Имя	Описание
Copy	Создает и возвращает полную копию объекта разрешения
Demand	Просматривает стек вызовов и генерирует исключение *Security Exception,* обнаружив, что какая-то функция была вызвана без наличия у вызывающей стороны соответствующих разрешений
Intersect	Создает и возвращает разрешение, которое предоставляет только права, общие для разрешений-аргументов
IsSubsetOf	Определяет, входит ли текущее разрешение в подмножество заданного разрешения
Union	Создает разрешение, которое является результатом объединения наборов прав, предоставляемых текущим и заданным разрешениями

В любых объектах разрешений в .NET Framework должен быть реализован интерфейс *IPermission,* предоставляющий общую функциональность всем объектам системы безопасности. Вызывая метод *Demand,* объекты разрешений инициируют проверку прав доступа. Этот метод запрещает вызов защищенного кода без соответствующих разрешений. Если требуемые разрешения отсутствуют, метод *Demand* генерирует исключение *SecurityException.* Метод *Copy* создает полную копию объекта разрешения, а метод *IsSubsetOf* проверяет, входит ли текущее разрешение в подмножество заданного. Методы *Union* и *Intersect* применяют для создания нового объекта разрешения на основе пары однотипных разрешений. Объект разрешения, возвращаемый методом *Union,* реализует все права, которые предоставляют разрешения-аргументы, а разрешение-результат метода *Intersect* — только права, общие для обоих разрешений, переданных методу *Intersect.* Об использовании этих методов для манипулирования объектами разрешений .NET Framework рассказано в следующих разделах.

Модель защиты, основанная на ролях

Система безопасности, основанная на ролях, предоставляет доступ к приложению (ресурсам) или отказывает в нем в зависимости от того, кем является пользователь и какова его роль в организации. Давайте рассмотрим приложение, которым пользуются менеджеры и рядовые сотрудники. В нем основная функциональность дос-

тупна любым пользователям, а доступ к конфиденциальным данным, таким, как платежные ведомости и персональная информация, — только менеджерам. Подобная система безопасности может быть реализована на основе ролей.

Примечание Все примеры кода из раздела, посвященного защите, основанной на ролях, предполагают наличие оператора *Imports System.Security.Principal* (Visual Basic .NET) или *using System.Security.Principal*(Visual C#).

Объект Principal

Объект *Principal* представляет в .NET Framework аутентифицированного пользователя. Этот объект содержит имя пользователя и его роль. Его можно сравнивать с объектом класса *PrincipalPermission*, который применяется для зашиты конфиденциальных ресурсов приложения от несанкционированного доступа.

В Windows-приложениях для проверки имени и роли пользователя применяются встроенные защитные механизмы Windows. Для этого нужно связать представляющий текущего пользователя объект *WindowsPrincipal* с приложением, установив политику *участника безопасности* (security principal) для текущего домена приложения, как показано ниже:

Visual Basic .NET

```
AppDomain.CurrentDomain.SetPrincipalPolicy _
   (PrincipalPolicy.WindowsPrincipal)
```

Visual C#

```
// Этот пример предполагает наличие оператора using System.Security.Principal.
AppDomain.CurrentDomain.SetPrincipalPolicy
   (PrincipalPolicy.WindowsPrincipal);
```

Объект *WindowsPrincipal* обладает ссылкой на объект *Windowsldentity,* представляющий текущего пользователя. Чтобы извлечь сведения о текущем пользователе, получите объект *Windowsldentity* через свойство *Identity* текущего объекта *WindowsPrincipal.* Объект *WindowsPrincipal* возвращает значение свойства *Identity* в виде интерфейса *IIdentity*, поэтому его следует явно привести к типу *Windowsldentity.*

Visual Basic .NET

```
' Этот пример предполагает, что политика участника безопасности
' установлена как WindowsPrincipal.
Dim myPrincipal As WindowsPrincipal
' Получить ссылку на текущий объект WindowsPrincipal.
myPrincipal = CType(Threading.Thread.CurrentPrincipal, _
   WindowsPrincipal)
Dim myIdentity As Windowsldentity
' Получить объект Windowsldentity для текущего участника безопасности.
myIdentity = CType(myPrincipal.Identity, Windowsldentity)
' Показать имя текущего пользователя.
MessageBox.Show(myIdentity.Name)
```

Visual C#

```
// Этот пример предполагает, что политика участника безопасности
// установлена как WindowsPrincipal.
WindowsPrincipal myPrincipal;
// Получить ссылку на текущий объект WindowsPrincipal.
myPrincipal = (WindowsPrincipal)
   System.Threading.Thread.CurrentPrincipal;
WindowsIdentity myIdentity;
// Получить объект WindowsIdentity для текущего участника безопасности.
myIdentity = (WindowsIdentity)myPrincipal.Identity;
// Показать имя текущего пользователя.
MessageBox.Show(myIdentity.Name);
```

Императивная защита, основанная на ролях

Класс *Principal Permission* также применяют для реализации императивной защиты. Объект *Principal Permission* представляет имя пользователя и его роль. Для успешной проверки прав доступа имя и роль текущего пользователя должны соответствовать удостоверениям, заданным объектом *Principal Permission*. Метод *Demand* объекта *Principal Permission* сравнивает заданные этим объектом имя и роль с аналогичными параметрами текущего объекта *Principal*. Следующий пример демонстрирует создание объекта *PrincipalPermission* и применение его для проверки текущего пользователя:

Visual Basic .NET

```
' Конструктор класса PrincipalPermission принимает
' в качестве параметров имя и роль пользователя.
Dim myPermission As New PrincipalPermission("Megan", "Manager")
' Проверить, определяет ли текущий объект Principal имя "Megan"
' и роль "manager".
myPermission.Demand()
```

Visual C#

```
// Конструктор класса PrincipalPermission принимает
// в качестве параметров имя и роль пользователя,
PrincipalPermission myPermission = new PrincipalPermission("Megan",
   "Manager");
// Проверить, определяет ли текущий объект Principal имя "Megan"
// и роль "manager".
myPermission.Demand();
```

Метод *Union* класса *PrincipalPermission* применяют для создания разрешения, предоставляющего все права, которые предоставляют два разрешения-аргумента этого метода. Вот как его вызывают:

Visual Basic .NET

```
Dim Permission1 As New PrincipalPermission("Megan", "Manager")
Dim Permission2 As New PrincipalPermission("Ann", "Group Manager")
```

```
Dim Permission3 As PrincipalPermission
' Объединить разрешения Permission1 и Permission2.
Permission3 = Permission2.Union(Permission1)
' Для доступа к этому коду текущий объект Principal
' должен представлять пользователя с именем Ann из группы Manager
' либо пользователя Megan из той же группы.
Permission3.Demand()
```

Visual C#

```
PrincipalPermission Permission1 - new PrincipalPermission("Megan", "Manager");
PrincipalPermission Permission2 - new PrincipalPermission("Ann", "Group Manager");
PrincipalPermission Permission3;
// Объединить разрешения Permission1 и Permission2.
Permission3 = (PrincipalPermission)Permission2.Union(Permission1);
// Для доступа к этому коду текущий объект Principal
// должен представлять пользователя с именем Ann из группы Manager
// либо пользователя Megan из той же группы,
Permission3.Demand();
```

Если при создании объекта *PrincipalPermission* передать *Nothing (null)* вместо имени или роли, создается разрешение, проверяющее соответственно только имя либо только роль. Следующий код создает объект *PrincipalPermission,* который проверяет только роль текущего объекта *Principal:*

Visual Basic .NET

```
' Создать разрешение, проверяющее только принадлежность
' текущего пользователя к менеджерам.
Dim myPermission As New PrincipalPermission(Nothing, "Manager")
```

Visual C#

```
// Создать разрешение, проверяющее только принадлежность
// текущего пользователя к менеджерам.
PrincipalPermission myPermission - new
   PrincipalPermission(null, "Manager");
```

Метод *Intersect* позволяет создать новое разрешение, которое предоставляет только права, общие для двух заданных разрешений:

Visual Basic .NET

```
Dim Permission1 As New PrincipalPermission(Nothing, "Manager")
Dim Permission2 As New PrincipalPermission("Megan", Nothing)
Dim Permission3 As PrincipalPermission
' Создать новое разрешение, предоставляющее только те права, которые
' являются общими для разрешений Permission1 и Permission2.
Permission3 = Permission2.Intersect(Permission1)
' Объект Principal, претендующий на доступ к этому коду, должен
' представлять пользователя с именем Megan и ролью Manager.
Permission3.Demand0
```

Visual C#

```
PrincipalPermission Permission1 = new PrincipalPermission(null, "Manager");
PrincipalPermission Permission2 = new PrincipalPermission("Megan", null);
PrincipalPermission Permission3;
// Создать новое разрешение, предоставляющее только те права, которые
// являются общими для разрешений Permission1 и Permission2.
Permission3 = (PrincipalPermission)Permission2.Intersect(Permission1);
// Объект Principal, претендующим на доступ к этому коду, должен
// представлять пользователя с именем Megan и ролью Manager.
Permission3.Demand();
```

Объекты *Principal Permissions* можно применять для проверки принадлежности к встроенным ролям Windows, таким, как Administrators. Встроенные роли необходимо задавать с префиксом BUILTIN\, как в следующем примере:

Visual Basic .NET

```
Dim myPermission as New PrincipalPermission("Rob", _
   "BUILTIN\Administrators")
```

Visual C#

```
// В программах на C# следует указывать двойной обратный слэш (\\),
// поскольку символ "\" задает escape-коды.
PrincipalPermission myPermission - new
   PrincipalPermission("Rob", "BUILTIN\\Administrators");
```

Декларативная защита, основанная на ролях

Любому объекту разрешения соответствует эквивалентный атрибут. Помечая такими атрибутами классы и члены, можно управлять доступом к ним. Атрибуты играют ключевую роль в *декларативной безопасности* (declarative security). При использовании этой модели системы безопасности защищаемые члены помечают атрибутами разрешений, определяющими соответствующий уровень доступа. Эти атрибуты также добавляют к метаданным типа, что позволяет администратору принять решение относительно исполнения сборки, проанализировав ее метаданные. Конструктор любого атрибута разрешения принимает параметр типа *SecurityAction*, определяющий действие, которое выполняет данный атрибут. В модели защиты, основанной на ролях, это, как правило, *Demand*. Свойства созданных атрибутов разрешений устанавливают при помощи оператора := (Visual Basic .NET) или = (Visual C#). Вот пример реализации для метода *myMethod* защиты, основанной на ролях:

Visual Basic .NET

```
<PrincipalPermission(SecurityAction.Demand, Name:="Joe", _
   Role:="Clerk")> Public Sub MyMethod()
' Реализация опущена.
End Sub
```

Visual C#

```
[PrincipalPermission(SecurityAction.Demand, Name="Joe",
   Role="Clerk")]
public void myMethod()
{
   // Реализация опущена.
}
```

Конфигурирование защиты по правам доступа к коду

Защита по правам доступа к коду призвана предотвратить использование кода приложения лицами, не имеющим на то полномочий, а также предоставить системному администратору доступ к требованиям, связанным с зашитой приложения. Обладая этой информацией, администратор сможет управлять исполнением конкретных сборок.

Подобно защите, основанной на ролях, зашита по правам доступа к коду базируется на разрешениях. Но если в первом случае разрешения представляют пользователей с соответствующими именами и/или ролями, то здесь разрешения представляют системные ресурсы и управляют доступом к ним. Рассмотрим для примера файловую систему. Если некоторое приложение записывает данные в файлы, необходимо защитить эти файлы от несанкционированного доступа, а файловую систему — от вреда, который способны причинить злоумышленники. Любой код, обращающийся к файловой системе, следует защитить объектом *File I O Permission*. Этот объект проверяет наличие необходимых разрешений у каждого, кто пытается вызвать защищенный им код.

Разрешения на доступ к коду

Каждое разрешение на доступ к коду защищает некоторый ресурс (таблица 9-3). Большинство подобных разрешений определено в пространстве имен *System Security*, за исключением немногих специальных, которые располагаются в других пространствах имен.

Таблица 9-3. Разрешения на доступ к коду

Имя	Описание
Directory Services Permission	Управляет доступом к Active Directory
EnvironmentPermission	Определяет возможность чтения и установки переменных окружения
EventLog Permission	Определяет возможность чтения и записи журнала событий
File DialogPermission	Управляет доступом к файлам и каталогам через диалоговое окно Open File
FileIOPermission	Определяет возможность создания, чтения и записи объектов файловой системы
OleDbPermission	Управляет доступом к базам данных OLEDB
PrintingPermission	Управляет доступом к принтеру
Reflection Permission	Управляет доступом к классам *System. Reflection c* целью получения сведений о типах во время выполнения

Таблица 9-3. *(окончание)*

Имя	Описание
Registry-Permission	Определяет возможность чтения и записи реестра
Security Permission	Управляет предоставлением ряда прав, в том числе на исполнение кода, манипулирование потоками и участниками безопасности, а также на вызов неуправляемого кода
SQLClientPermission	Управляет доступом к базам данных Microsoft SQL Server
UIPermission	Управляет доступом к пользовательскому интерфейсу

Этот, далеко не полный, список позволяет, тем не менее, составить представление о разрешениях, применяемых для зашиты приложений.

Создание разрешений на доступ к коду

Каждое разрешение на доступ к коду поддерживает собственный набор перегруженных конструкторов, позволяющий по-разному конфигурировать защищаемые им ресурсы. Можно создать разрешение, управляющее доступом ко всем защищаемым им ресурсам либо к их подмножеству. Для этого применяют соответствующие параметры во время создания объекта разрешения.

Допускается создавать разрешения, предоставляющие неограниченный доступ к ресурсам, при помощи флага *PermissionState. Unrestricted.* Напротив, флаг *PermissionState.None* позволяет создать разрешение, которое вовсе не дает доступа к защищаемым ресурсам. Вот как это делается:

Visual Basic .NET

```
' Предоставить неограниченный доступ к ресурсам отражения.
Dim myPermission As New _
   ReflectionPermission(PermissionState.Unrestricted)
' Полностью запретить доступ к ресурсам пользовательского интерфейса.
Dim anotherPermission As New _
   UIPermission(PermissionState.None)
```

Visual C#

```
// Предоставить неограниченный доступ к ресурсам отражения.
ReflectionPermission myPermission = new
   ReflectionPermission(PermissionState.Unrestricted);
// Полностью запретить доступ к ресурсам пользовательского интерфейса.
UIPermission anotherPermission = new
   UIPermission(PermissionState.None);
```

Каждое разрешение поддерживает дополнительные конструкторы, позволяющие создавать по-разному настроенные объекты разрешений. Подробное обсуждение всех версий конструкторов выходит за рамки этого учебного курса, но разработчику полезно знать о них и о предоставляемой ими возможности «тонкой настройки» разрешений. Вот как создать разрешение, управляющее доступом для записи к единственному файлу файловой системы:

Visual Basic .NET

```
Dim myPermission As New _
    FileIOPermission(FileIOPermissionAccess.Write, "C:\myFile.txt")
```

Visual C#

```
FileIOPermission myPermission = new
    FileIOPermission(FileIOPermissionAccess.Write, "C:\\myFile.txt");
```

Члены класса CodeAccessPermission

Все объекты разрешений на доступ к коду происходят от класса *CodeAccessPermission,* поэтому все они поддерживают одни и те же методы для проверки прав доступа и реализации политики безопасности. Самые важные из них описаны в таблице 9-4.

Таблица 9-4. Методы класса *CodeAccessPermission*

Метод	Описание
Assert	Позволяет вызывающему этот метод коду получить доступ к ресурсам, защищенным некоторым разрешением, даже если код, расположенный выше по стеку вызовов, не обладает необходимыми для этого разрешениями
Demand	Требует наличия у всех функций, расположенных выше по стеку вызовов, разрешения на доступ к защищенному данным разрешением ресурсу
Deny	Отказывает коду, вызвавшему этот метод, в доступе к ресурсу, защищенному данным разрешением
PermitOnly	Отказывает коду, вызвавшему этот метод, в доступе ко *всем* ресурсам, защищенным данным разрешением, за исключением заданного подмножества этих ресурсов
RevertAll	Отменяет все разрешения, предоставленные или отобранные ранее с помощью методов *Assert*, *Deny* и *PermitOnly*
RevertAssert	Удаляет все разрешения, предоставленные ранее при помощи метода *Assert*
Revert Deny	Возвращает все разрешения, отозванные ранее при помощи метода *Deny*
Revert Permit Only	Удаляет все разрешения, предоставленные или отобранные ранее с помощью метода *PermitOnly*

В следующем разделе обсуждается применение этих методов для реализации защиты по правам доступа к коду.

Императивная защита по правам доступа к коду

Подобно защите, основанной на ролях, защита по правам доступа к коду может быть императивной или декларативной. При использовании императивной защиты по правам доступа к коду запрос и проверка разрешений производится во время выполнения.

Метод *Demand* является ключевым для защиты по правам доступа к коду. CLR предоставляет разрешение на доступ к защищенным ресурсам по результатам про-

верки политики безопасности, установленной системным администратором для данной сборки. При вызове метода *Demand* некоторого объекта разрешения происходит следующее: метод *Demand* просматривает стек, проверяя у каждой вызывающей процедуры, расположенной выше в стеке вызовов, наличие разрешения на доступ к ресурсу, защищенному объектом разрешения, чей метод *Demand* вызван. Так, метод доверяемой сборки может вызывать другой метод, защищенный соответствующим разрешением. Если защищенный метод вызван доверяемой сборкой, вызов будет успешным и доступ к защищенному ресурсу будет предоставлен. Если сборка, с которой не установлены доверительные отношения, вызывает метод из доверяемой сборки, а тот, в свою очередь, вызывает защищенный метод, происходит следующее, Объект разрешения просматривает стек вызовов, чтобы убедиться в наличии разрешений на доступ к защищенному ресурсу у всех «звеньев» цепочки вызовов. Первая сборка не имеет соответствующего разрешения, поэтому вызов заканчивается неудачей.

Вот пример применения метода *Demand:*

Visual Basic .NET

```
' Создать объект разрешения, предоставляющего
' неограниченный доступ к файловой системе.
Dim myPermission As New _
   FileIOPermission(PermissionState.Unrestricted)
' У всех процедур, вызывающих данный код, должно быть
' разрешение на неограниченный доступ к файловой системе,
myPermission.Demand()
```

Visual C#

```
// Создать объект разрешения, предоставляющего
// неограниченный доступ к файловой системе.
FileIOPermission myPermission = new
   FileIOPermission(PermissionState.Unrestricted);
// У всех процедур, вызывающих данный код, должно быть
// разрешение на неограниченный доступ к файловой системе,
myPermission.Demand();
```

Разрешения на доступ кода годятся и для организации защиты кода на уровнях, расположенных выше и ниже заданной политики безопасности. Например, метод *Deny* запрещает доступ к ресурсу, защищенному данным объектом разрешения, даже если CLR предоставила соответствующее разрешение вызывающей стороне:

Visual Basic .NET

```
' Создать объект разрешения, предоставляющего
' неограниченный доступ к файловой системе.
Dim myPermission As New _
   FileIOPermission(PermissionState.Unrestricted)
' Отказывает текущему методу в доступе к файловой системе.
myPermission.Deny()
```

Visual C#

```
// Создать объект разрешения, предоставляющего
// неограниченный доступ к файловой системе.
FileIOPermission myPermission = new
   FileIOPermission(PermissionState.Unrestricted);
// Отказывает текущему методу в доступе к файловой системе.
myPermission.Deny();
```

Сходным образом действует метод *PermitOnly*. Он позволяет отказать в доступе ко всем ресурсам категории, защищенной некоторым объектом разрешения, предоставив его только для подмножества ресурсов этой категории, явно связанного с объектом разрешения. Предположим, что был создан объект *FileIOPermission*, управляющий доступом для записи к некоторому файлу. В результате вызова метода *PermitOnly* данного объекта разрешения любые попытки доступа к файловой системе окончатся неудачей, кроме попыток записи в указанный файл. Вот пример применения метода *PermitOnly*,

Visual Basic .NET

```
' Создать объект разрешения на запись в указанный файл,
Dim myPermission As Mew _
   FileIOPermission(FileIOPermissionAccess.Write, "C:\myFile.txt")
' Разрешается лишь запись в указанный файл, остальные виды
' доступа к файловой системе запрещаются.
myPermission.PermitOnly()
```

Visual C#

```
/7 Создать объект разрешения на запись в указанный файл.
FileIOPermissionmyPermission=new
   FileIOPermission(FileIOPermissionAccess.Write, "C:\\myFile.txt");
// Разрешается лишь запись в указанный файл, остальные виды
// доступа к файловой системе запрещаются.
myPermission.PermitOnly();
```

Метод *Assert* объявляет, что вызвавший его метод обладает разрешением на доступ к запрошенному ресурсу. Вызов метода *Assert* прекращает просмотр стека методом *Demand* с целью проверки наличия данного разрешения. Так, если сборка, которая не состоит в доверительных отношениях, вызовет метод, который, в свою очередь, вызовет *Assert*, а затем метод, защищенный методом *Demand*, произойдет следующее: инициированная методом *Demand* проверка будет остановлена, а исполнение вызова — продолжено, даже если в начале цепочки вызовов будет сборка, не пользующаяся доверием. Метод *Assert* таит потенциальную опасность, поскольку может предоставить доступ к защищенным ресурсам сборке, с которой не установлены доверительные отношения. Используйте метод *Assert* осмотрительно, не вызывайте его в обход системной политики безопасности. Для корректного вызова *Assert* необходимо предоставить сборке соответствующие разрешения:

Visual Basic .NET

```
' Создать объект разрешения, предоставляющего
' неограниченный доступ к файловой системе.
Dim myPermission As New _
   FileIOPermission(PermissionState.Unrestricted)
' Предоставить разрешение на доступ к файловой системе.
myPermission.Assert()
```

Visual C#

```
// Создать объект разрешения, предоставляющего
// неограниченный доступ к файловой системе.
FileIOPermission myPermission = new
   FileIOPermission(PermissionState.Unrestricted);
// Объявляется разрешение на доступ к файловой системе
myPermission.Assert();
```

Методы *Revert,* поддерживаемые всеми классами разрешений, отменяют действие методов *Deny*, *Assert* или *PermitOnly*. Это статические методы: они вызываются по имени класса, а не объекта, и действуют на все объекты данного класса. Вот пример применения методов *Revert*

Visual Basic .NET

```
' Отменить все модификации объекта ReflectionPermission.
ReflectionPermission.RevertAll()
' Отменить действие метода Deny на объект EnvironmentPermission.
EnvironmentPermission.RevertDeny()
' Отменить действие метода Assert на объект FileIOPermission.
FileIOPermission.RevertAssert()
' Отменить действие метода PermitOnly на объект MessageQueuePermission.
MessageQueuePermission.RevertPermitOnly()
```

Visual C#

```
// Отменить все модификации объекта ReflectionPermission.
ReflectionPermission.RevertAll();
// Отменить действие метода Deny на объект EnvironmentPermission.
EnvironmentPermission.RevertDeny();
// Отменить действие метода Assert на объект FileIOPermission.
FileIOPermission.RevertAssert();
// Отменить действие метода PermitOnly на объект MessageQueuePermission.
MessageQueuePermission.RevertPermitOnly();
```

Декларативная защита по правам доступа к коду

Для управления доступом к системным ресурсам можно использовать декларативную защиту по правам доступа к коду вместо императивной защиты. Как и в случае защиты, основанной на ролях, у каждого разрешения на доступ кода имеется эквивалентный атрибут, которым можно пометить метод или класс. Декларативная за-

шита по правам доступа к коду годится также для проверки разрешений сборки в целом.

Процедура применения атрибутов разрешении доступа к коду с классами и методами не отличается от таковой для атрибутов защиты, основанной на ролях, за исключением одного. Вместо роли в данном случае требуется указать значение *SecurityAction*, которое представляет действие, предпринимаемое данным атрибутом. Ниже показано, как при помощи декларативной безопасности отказать в предоставлении разрешения *FileIOPermission* для класса:

Visual Basic .NET

```
<FileIOPermission(SecurityAction.Deny)>Public Class aClass
    ' Реализация опущена.
End Class
```

Visual C#

```
[FileIOPermission(SecurityAction.Deny)]
public class aClass
{
   // Реализация опущена.
}
```

Флаги *SecurityAction. Demand*, *SecurityAction. Deny*, *SecurityAction.Assert* и *SecurityAction.PermitOnly* эквивалентны методам *Demand*, *Deny*, *Assert* и *PermitOnly* соответствующих классов разрешений. Декларативная модель защиты поддерживает ряд дополнительных средств защиты, которые применяются к классам и методам. Флаг *SecurityAction.LinkDemand* требует наличия права доступа к классу или методу только у процедуры, непосредственно вызывающей этот класс или метод, а флаг *SecurityAction. InheritanceDemand* — наличия соответствующих прав у любого класса, производного от защищенного класса либо переопределяющего защищенный метод.

Атрибуты разрешений позволяют задавать требования безопасности и для сборки в целом. Существует три флага *SecurityAction*, которые можно определять в директивах, действующих на уровне сборки. Флаг *SecurityAction. RequestMinimum* просит CLR назначить сборке заданное разрешение. Если системная политика безопасности запрещает предоставление запрошенного разрешения, сборка не будет исполнена. Флаг *SecurityAction. RequestOptional* напоминает предыдущий, но позволяет исполнить сборку даже при отказе в предоставлении запрошенных им разрешений. Флаг *SecurityAction. RequestRefuse* заставляет CLR отказать сборке в предоставлении указанных им прав. Все эти флаги определяют внутри директивы *Assembly (assembly):*

Visual Basic .NET

```
<Assembly: FileIOPermission(SecurityAction.RequestMinimum)>
```

Visual C#

```
[assembly: FileIOPermission(SecurityAction.RequestMinimum)]
```

Как и при использовании декларативной защиты на основе ролей, атрибуты разрешений допускается инициализировать любыми разрешениями при помощи опе-

ратора := (Visual Basic .NET) или = (Visual C#). Следующий пример демонстрирует декларативное предоставление разрешения на доступ к файлу:

Visual Basic .NET

```
<FileIOPermission(SecurityAction.Assert, _
   Write:="C:\myFile.txt")>Public Sub WriteFile()
    ' Реализация опущена.
End Sub
```

Visual C#

```
[FileIOPermission(SecurityAction.Assert,
   Write="C:\\myFile.txt")]
public void WriteFileO
{
   // Реализация опущена.
}
```

Обработка исключений и императивная защита

Приложение должно отслеживать возникновение ситуаций, чреватых ошибками, и адекватно обрабатывать любые исключении. Нарушение защиты приводит к генерации исключения *SecurityException*, поэтому любые запросы разрешений при использовании императивной защиты следует помещать в блоки обработки исключений. Это позволит избежать немедленного завершения приложения в случае отказа на предоставление запрошенных разрешений и попытаться запросить разрешения в меньшем объеме. Разработчик должен сам выбрать подходящее действие, которое следует предпринять в ответ на отказ в предоставлении того или иного разрешения. Например, можно разрешить пользователю сохранить данные и штатно завершить работу приложения либо продолжить работу без запрошенного ресурса. Так или иначе, требуется предусмотреть обработку любых исключений, которые только можно предвидеть.

Резюме

- Существует два основных типа защиты кода: защита, основанная на ролях, использует аутентификацию пользователей и их ролей, а защита по правам доступа к коду предотвращает несанкционированный доступ к системным ресурсам.
- Возможности доступа к коду и системным ресурсам контролируют объекты разрешений. В каждом из них реализован интерфейс *IPermission*, предоставляющий базовый набор функций.
- Авторизация на основе ролей проверяет роль и имя пользователя, представленного текущим объектом *Principal*. Применение встроенных механизмов аутентификации Windows позволяет сверять этот объект с объектом *WindowsPrincipal*. Для этого необходимо установить для текущего домена приложения (*AppDomain*) соответствующую политику участника безопасности (*PrincipaPolicy*).

- Возможности доступа к системным ресурсам контролирует объект разрешения доступа к коду. Такой объект способен представить полный доступ к связанной с ним категории ресурсов либо к заданному подмножеству ресурсов данного типа.
- Запрос разрешений осуществляется императивно либо декларативно. Декларативную защиту реализуют добавлением к классам и методам атрибутов разрешений с заданными действиями защиты. Императивную защиту реализуют через соответствующие методы объекта разрешения, который представляет объект участника безопасности (при использовании защиты, основанной на ролях) или объект разрешения на доступ к системному ресурсу (при использовании защиты по правам доступа к коду).
- Декларативная защита позволяет запрашивать разрешения для сборки в целом. Для этого применяются соответствующие флаги, объявленные внутри директивы *Assembly* (*assembly*).

Практикум 9. Конфигурирование и защита приложения

Задача этого практикума — усовершенствовать приложение, созданное в практикуме 8. Вы добавите к нему код, который позволит назначать культуру через конфигурационный файл, затем дополните приложение системой безопасности, основанной на ролях, чтобы защитить его от несанкционированного доступа. В завершение вы организуете в приложении защиту по правам доступа к коду, чтобы запретить несанкционированную печать данных. Решение задач этого практикума вы найдете в каталоге \Labs\Ch09\Solution на прилагаемом к книге компакт-диске.

Прежде всего

Для выполнения этого практикума вам потребуются результаты практикума 8 (готовое решение этого практикума вы можете взять с прилагаемого к книге компакт-диска). Чтобы успешно выполнить все упражнения, необходимо быть членом группы администраторов Windows на компьютере, на котором вы будете работать.

Продолжительность практикума - около 45 минут.

Упражнение 1. Создание конфигурационного файла

Сейчас вы создадите конфигурационный файл для приложения и добавите к нему код, читающий из него значение текущей культуры и устанавливающий соответствующие значения для свойств *CurrentCulture* и *CurrentUICulture*. Прежде чем приступать к этому упражнению, загрузите в Visual Studio .NET решение, созданное в практикуме 8.

▶ Создание конфигурационного файла

1. В Solution Explorer щелкните правой кнопкой форму *frmLanguage* и выберите команду Delete. В ответ на вопрос щелкните OK — форма *frmLanguage* удаляется из проекта.
2. В Solution Explorer щелкните правой кнопкой форму *Form1* и выберите команду View Code — откроется окно кода этой формы.
3. Найдите конструктор *Form1* и удалите из него следующие строки:

Visual Basic .NET

```
Dim aForm As New frmLanguage()
aForm.ShowDialog()
```

Visual C#

```
frmLanguage aForm = new frmLanguage();
aForm.ShowDialog();
```

4. Добавьте к приложению конфигурационный файл. В зависимости от языка программирования это делается по-разному.

Visual Basic .NET

1. В меню Project выберите Add New Item.
2. В окне Add New Item щелкните Application Configuration File — к проекту будет добавлен новый конфигурационный файл.

Visual C#

1. В меню Project выберите Add New Item.
2. В окне Add New Item щелкните Text File — к приложению добавляется текстовый файл. Новый файл открывается в текстовом редакторе.
3. В Solution Explorer щелкните правой кнопкой созданный текстовый файл и выберите команду Rename, чтобы переименовать его в App.config. Откройте это файл в текстовом редакторе и добавьте в него следующий XML-код:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

В Solution Explorer дважды щелкните файл App.config. Система спросит, нужно ли закрыть файл, — щелкните кнопку Yes (Да). На экране отобразится файл App.config в редакторе XML.

5. Добавьте к элементу <configuration> файла App.config следующий XML-код.:

```
<appSettings>
    <!-- Замените "Culture" кодом текущей культуры -->
    <!-- Замените "UICulture" кодом текущей культуры
         пользовательского интерфейса -->
    <add key="Culture" value="it-IT" />
    <add key="UICulture" value="it-IT" />
</appSettings>
```

6. В окне Solution Explorer щелкните правой кнопкой форму *Form1* и выберите команду View Code — код этой формы откроется в редакторе.
7. Добавьте в конструктор *Forml* код, который будет читать значения из конфигурационного файла и назначать их свойствам *CurrentCulture* и *CurrentUlCulture*.

Visual Basic .NET

```
Dim Reader As New System.Configuration.AppSettingsReader()
Threading.Thread.CurrentThread.CurrentCulture = New _
   Globalization.CultureInfo(CType(Reader.GetValue("Culture", _
   GetType(String)), String))
Threading.Thread.CurrentThread.CurrentUICulture = New _
   Globalization.CultureInfo(CType(Reader.GetValue("UICulture", _
   GetType(String)), String))
```

Visual C#

```
System.Configuration.AppSettingsReader reader = new
   System.Configuration.AppSettingsReader();
System.Threading.Thread.CurrentThread.CurrentCulture = new
```

```
    System.Globalization.CultureInfo((string)
    (reader.GetValue("Culture", typeof(string))));
System.Threading.Thread.CurrentThread.CurrentUICulture = new
    System.Globalization.CultureInfo((st.ring)
    (reader.GetValue("UICulture", typeof(string))));
```

8. Нажмите F8, чтобы протестировать приложение. Когда откроется Forml, обратите внимание, что пользовательский интерфейс отображается на итальянском языке, а в списке для выбора валюты присутствует знак евро.

Упражнение 2. Защита приложения

Сейчас вы добавите к приложению код, необходимый для его защиты. Для проверки роли текущего пользователя вы примените императивную защиту, а для защиты доступа к принтеру — декларативную.

▶ Как защитить приложение

1. В Solution Explorer щелкните правой кнопкой *Forml* и выберите View Code — код формы Form] откроется в редакторе.
2. Добавьте к конструктору формы Forml код, ограничивающий круг лиц, имеющих доступ к приложению, членами встроенной группы Windows Administrators:

Visual Basic .NET

```
AppDomain.CurrentDomain.SetPrincipalPolicy _
    (System.Security.Principal.PrincipalPolicy.WindowsPrincipal)
Dim myPerm As New _
    System.Security.Permissions.PrincipalPermission(Nothing, _
    "BUILTIN\Administrators")
Try
        myPerm. Demand0
Catch se As System.Security.SecurityException
    MessageBox.Show _
        ("You do not have permission to run this program!")
    End
End Try
```

Visual C#

```
AppDomain.CurrentDomain.SetPrincipal Policy
    (System.Security.Principal.PrincipalPolicy.WindowsPrincipal);
System.Security.Permissions.PrincipalPermission myPerm - new
    System.Security.Permissions.PrincipalPermission(null,
    "BUILTIN\\Administrators");
try
{
        myPerm. Demand0;
    }
catch(System.Security.SecurityException se)
```

```
{
   MessageBox.Show("You do not have permission to run this program!");
   throw se;
}
```

3. Найдите метод *btnPrint_Click* и добавьте к нему следующий атрибут. Он необходим, чтобы только доверенный код смог выводить данные на принтер:

Visual Basic .NET

```
<Drawing.Printing.PrintingPermission _
   (Security.Permissions.SecurityAction.Demand, _
   Level:=Drawing.Printing.PrintingPermissionLevel.AllPrinting)>
```

Visual C#

```
[System.Drawing.Printing.PrintingPermission
   (System.Security,Permissions.SecurityAction.Demand,
   Level=System.Drawing.Printing.PrintingPermissionLevel.AllPrinting)]
```

4. Добавьте к методу *btnPrint_Click* обработку исключений, которые могут возникнуть при попытке вывода на печать:

Visual Basic .NET

```
Try
   PrintDocument1.Print()
Catch
   MessageBox.Show("You do not have permission to print!")
End Try
```

Visual C#

```
try
{
   printDocument1.Print();
}
catch
{
   MessageBox.Show("You do not have permission to print!");
}
```

5. Нажмите F5, чтобы протестировать приложение — оно должно работать нормально.

Примечание Если вы не входите в группу Administrators, приложение работать не будет. В этом случае задайте роль, соответствующую встроенной группе Windows, членом которой вы являетесь. Если при попытке печати возникает исключение безопасности, настройте для сборки приложения соответствующую политику безопасности самостоятельно или попросить об этом системного администратора.

---

6. В строке, где создается объект *PrincipalPermission,* замените BUILTIN\Administrators на любую роль, которая не существует в Windows (например, на BUILTIN\Administratorss). Нажмите F4, чтобы скомпоновать и запустить приложение, Приложение отвергнет запрошенное вами разрешение и завершится неудачей. В программе на Visual Basic .NET приложение просто завершится, а программа на Visual C# генерирует необработанное исключение, которое генерируется повторно. Верните роли прежнее значение (BUILTIN\Administrators).
7. В атрибуте разрешения *PrintingPermission* замените действие *Demand* на *Deny.* Нажмите F5, чтобы скомпоновать и запустить приложение. Попробуйте что-нибудь напечатать.

   Программа отказывает в предоставлении запрошенного разрешения и уведомляет о этом пользователя. Верните исходное значение флагу действия защиты и перекомпилируйте приложение.
8. В меню File выберите команду Save All. чтобы сохранить свою работу.

Закрепление материала

Приведенные ниже вопросы помогут вам лучше усвоить основные темы данной главы. Если вы не сумеете ответить на вопрос, повторите материал соответствующего занятия. Ответы для самопроверки — в приложении «Вопросы и ответы» в конце книги.

1. Как подписать сборку с помощью строгого имени и для чего это нужно?
2. Расскажите, как извлечь ресурсы во время выполнения.
3. Как во время выполнения извлечь информацию из конфигурационного файла? А как записать в него нужные данные во время разработки?
4. Вы создали решение, которое должно быть доступным только членам групп Developers и Administrators локального компьютера. Как реализовать такую схему безопасности?
5. Кратко перечислите различия между декларативной и императивной моделями при использовании защиты по правам доступа к коду.

ГЛАВА 10

Развертывание

Занятие 1. Планирование развертывания приложения 434

Занятие 2. Конфигурирование проекта установочной программы 446

В этой главе

В этой главе рассказано, как планировать и выполнять развертывание приложений через локальную сеть, Интернет, а также со съемных носителей. Кроме того, вы узнаете, как создать проект установочной программы для своего приложения и применить ее для развертывания на локальном компьютере.

Прежде всего

Для изучения материалов этой главы вам необходимо изучить главы 1–9.

Занятие 1. Планирование развертывания приложения

Чтобы приложение оценили по достоинству, оно прежде всего должно оказаться на компьютерах своей целевой аудитории. Microsoft Visual Studio .NET поддерживает самые разные способы развертывания проектов: от простейшего с использованием команды XCOPY до самого сложного и гибкого с применением программы Microsoft Windows Installer. На этом занятии вы научитесь подбирать оптимальный способ развертывания для своего приложения.

Изучив материал этого занятия, вы сможете:

- ✓ рассказать о развертывании приложений Windows Forms с применением XCOPY;
- ✓ создать и настроить проект установочной программы для приложения с помощью мастера Setup Project;
- ✓ спланировать развертывание приложения со сменных носителей;
- ✓ спланировать развертывание приложения через сеть;
- ✓ спланировать развертывание приложения с Web-сервера.

Продолжительность занятия - около 45 минут,

Итак, код приложения написан и отлажен, приложение полностью готово к развертыванию, цель которого — установить его и необходимые сопутствующие файлы на компьютеры клиентов, затратив минимальные усилия.

Простые приложения .NET Framework позволяет развертывать, просто копируя их каталоги на клиентские компьютеры. Для развертывания более сложных приложений в Visual Studio .NET предусмотрена технология Windows Installer, позволяющая создавать проекты установочных программ с широкими возможностями настройки.

Развертывание приложений с помощью XCOPY

В развертывании с помощью XCOPY нет ничего замысловатого. Команда XCOPY получила свое название от одноименной команды DOS, копирующей каталог со всем его содержимым в заданный каталог. В некоторых случаях для развертывания приложений достаточно и этого простого метода.

Развертывание при помощи XCOPY выполняется через командную строку. При вызове команды XCOPY необходимо задать исходный и целевой каталоги; параметр /s заставляет скопировать исходный каталог со всеми вложенными каталогами. Так, чтобы скопировать каталог MyApplication со всеми вложенными в него каталогами с диска D: на диск C:, наберите в командной строке следующую команду;

```
XCOPY D:\MyApplication C:\MyApplication /s
```

Чтобы получить сведения о других параметрах, наберите в командной строке HELP XCOPY.

Однако этот способ имеет ряд ограничений. Во-первых, все необходимые приложению файлы должны располагаться в каталоге приложения, включая исполняемые файлы и библиотеки с компонентами и элементами управления, а также все ресурсы и другие необходимые приложению файлы. Во-вторых, до начала развер-

тывания на целевых компьютерах следует установить инфраструктуру .NET Framework. В-третьих, развертываемое таким образом приложение не должно требовать иных файлов или ресурсов (например баз данных или разделяемых компонентов), кроме тех, что уже установлены на целевом компьютере на момент развертывания. Как правило, XCOPY применяют для развертывания приложений, для работы которых годится лишь .NET Framework, и если заранее известно, что эта инфраструктура установлена на всех целевых машинах.

▶ **Развертывание приложений с помощью XCOPY**

1. Убедитесь, что приложение отвечает всем указанным требованиям.
2. Вызовите окно командной строки. В Windows XP для этого нужно щелкнуть Start\All Programms\Accessories (Пуск\Все программы\Стандартные).
3. Вызовите из командной строки XCOPY, указав исходный и целевой каталоги и все необходимые параметры, например:

```
XCOPY D:\MyApplication C:\MyApplication /s
```

Создание проекта установочной программы

Если возможностей XCOPY недостаточно для развертывания приложений, можно воспользоваться поддержкой Visual Studio .NET и создать проект установочной программы, использующей пакет Windows Installer. Так удается создать весьма гибкие инструменты для установки приложений, поддерживающие самые разные сценарии развертывания. Чтобы создать проект установочной программы для приложения, просто добавьте к существующему решению специальный проект.

Для Windows Forms поддерживается два вида подобных проектов: проекты *установочных программ для приложений* (Setup project) и проекты *для дополнительных модулей* (Merge module projects). Первые применяют для развертывания исполняемых приложений. После компиляции такого проекта получается дистрибутив, пригодный для установки приложения на целевой компьютер. Вторые используются для развертывания элементов управления и компонентов, которые не являются отдельными приложениями и не подлежат непосредственному развертыванию. В этой форме распространяются пакеты обновлений, которые разрешается объединять с имеющимся проектом установочной программы. Проект установочной программы для дополнительного модуля содержит все необходимые сведения о том, где я как следует установить некоторую сборку, но работает он только вместе с определенным приложением.

Мастер Setup Project

Мастер Setup Project из Visual Studio .NET позволяет выбрать для вашего проекта подходящий тип установочной программы. Направляя действия пользователя, он позволяет быстро создать заготовку проекта установочной программы, которую затем «доводят» с помощью Visual Studio .NET.

Чтобы добавить к решению проект установочной программы, выберите в меню File команду Add Project\New Project — откроется диалоговое окно Add New Project. Далее вызовите мастер Setup Project (рис. 10-1) — для этого на панели Project Types выберите Setup and Deployment Projects, а в окне Templates — Setup Wizard.

Щелкните кнопку Next, чтобы открыть страницу Choose a Project Type (рис. 10-2), на которой задают тип проекта. Если нужно создать проект установочной программы для приложения, выберите Create A Setup For A Windows Application, а если для

установки DLL-файла (то есть для установки неисполняемой сборки), укажите Create a Merge Module For Windows Installer. Щелкните кнопку Next.

Рис. 10-1. Мастер Setup Project

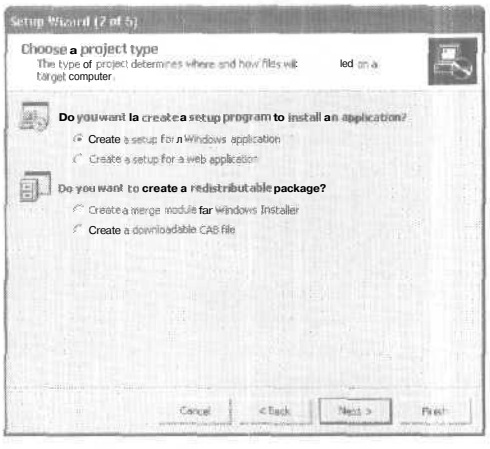


Рис. 10-2. Выбор типа проекта

Страница Choose Project Outputs To Include позволяет выбрать файлы решения для включения в проект установочной программы. Обычно на этой странице для каждого проекта решения отображается пять флажков (это для приложений на Visual Basic .NET, в случае Visual C# их шесть) — по одному для каждого из типов содержимого, добавляемого к проекту. Если щелкнуть какой-либо значок, в поле Description появится его описание. На рис. 10-3 показан вариант страницы Choose Project Outputs To Include для Visual Basic .NET.

Чтобы включить в проект EXE- и DLL-файлы, установите флажок Primary Output. При развертывании приложения для тестирования удобно также добавлять файлы с исходным текстом и символами для отладки, но в окончательной версии дистрибутива их быть не должно. Щелкните кнопку Next, чтобы перейти на следующую страницу.

Страница Choose Files To Include (рис. 10-4) позволяет добавить к проекту любые дополнительные файлы, например текстовые файлы readme, HTML-страницы

со справочной информацией, а также файлы поддержки, которые обычно не включают в установочный комплект приложения. Чтобы выбрать эти файлы, щелкните кнопку Add и найдите нужный файл, Закончив, щелкните Next.

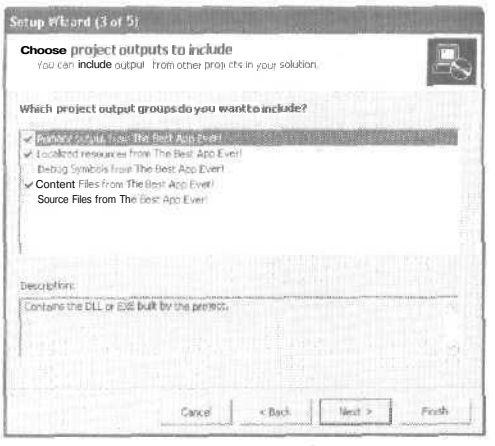


Рис. 10-3. Выбор файлов, добавляемых в проект установочной программы

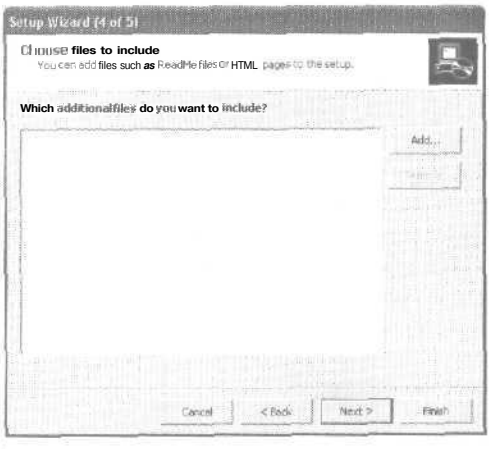


Рис. 10-4. Выбор дополнительных файлов для добавления к проекту установочной программы

Страница Create Project (рис. 10-5) отображает подробные сведения о параметрах, заданных в ходе работы с мастером. Щелкните кнопку Finish, чтобы создать проект установочной программы и добавить его к решению. К готовому проекту можно в любое время добавить дополнительные файлы. Для этого щелкните его правой кнопкой в окне Solution Explorer и выберите в контекстном меню Add подходящий элемент.

▶ **Создание проекта установочной программы с помощью мастера Setup Project**

1. В меню File выберите команду Add Project\New Project — откроется диалоговое окно Add New Project.
2. На панели Project Types выберите Setup and Deployment Projects, а на панели Templates — Setup Wizard.

3. Укажите в списке под приветствием тип проекта: для DLL это Merge Module, а для приложения — Setup For A Windows Application.
4. На следующих двух страницах задайте файлы проекта приложения и дополнительные файлы, которые следует включить в проект установочной программы.
5. На итоговой странице щелкните кнопку Finish, чтобы создать проект.

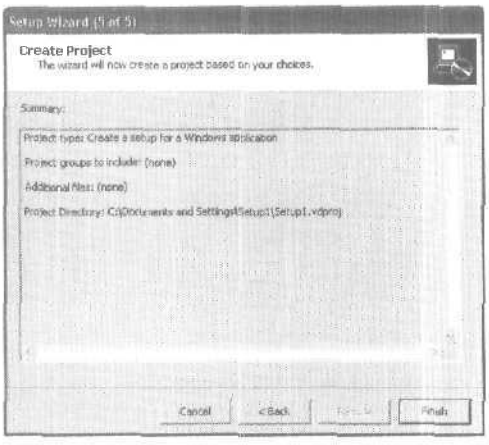


Рис. 10-5. Итоговая страница мастера Setup Project

Новый проект будет добавлен к решению и отобразится в окне Solution Explorer (рис. 10-6). В дополнение к выбранным вами файлам Visual Studio .NET автоматически найдет возможные все зависимости для нового проекта и добавит их в каталог Detected Dependencies. Например, на рис. 10-6 показана зависимость от пакета .NET Framework. Учтите, что зависимости, не добавленные к проекту установочной программы явно, по умолчанию исключаются из сборки. Чтобы добавить найденную зависимость к проекту, щелкните правой кнопкой соответствующий файл в окне Solution Explorer и в раскрывающемся списке сбросьте флажок Exclude для этого файла. Обратите внимание: включение установочных файлов .NET Framework существенно увеличивает размер дистрибутива, поэтому добавлять их стоит только при отсутствии .NET Framework на целевом компьютере.

Рис. 10-6. Найденная зависимость

Параметры компоновки установочной программы

Добавив к решению проект установочной программы, можно задать способ его компиляции, настроив параметры компоновки этого проекта. Обычно в результате компиляции установочного проекта получается по крайней мере один файл, используемый программой Windows Installer (.msi-файл), в котором хранятся все конфигурационные данные и содержимое, необходимое для установки приложения на целевой компьютер. Однако в зависимости от параметров компоновки разрешается заключать часть содержимого пакета в отдельные файлы, а также добавлять к нему установочные файлы Windows Installer, если он еще не установлен на целевом компьютере. Чтобы вызвать окно параметров компоновки проекта установочной программы, в Solution Explorer щелкните этот проект правой кнопкой и выберите из контекстного меню команду Properties — откроется диалоговое окно свойств проекта (рис. 10-7),

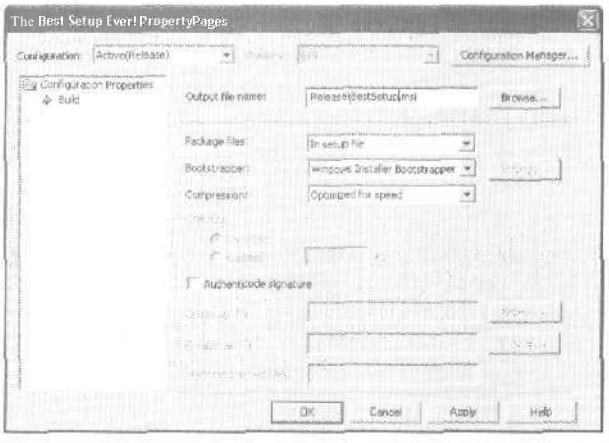


Рис. 10-7. Диалоговое окно свойств проекта установочной программы

Параметры компоновки (рис. 10-7) определяют результат компиляции проекта установочной программы, их набор зависит от способа развертывания приложения. Более подробно об этих параметрах рассказывается в следующих разделах.

Output File Name

Параметр Output File Name определяет имя, которое будет назначено выходному файлу, предназначенному для использования программой Windows Installer, а также каталог, в который он будет записан. По умолчанию его значение имеет вид *<configuration>\<projectname>.<extension>*, где *<configuration>* — каталог в структуре каталогов проекта; *<projectname>* (по умолчанию) — имя проекта, а *<extension>* — .msi для приложений Windows Installer или .msm — для дополнительных модулей Windows Installer. Чтобы изменить путь выходного файла, щелкните кнопку Browse и укажите новый каталог.

Package Files

Этот параметр определяет способ упаковки выходных файлов решения в дистрибутиве приложения. По умолчанию все выходные файлы упаковываются в один *файл* вместе с установочной программой. Это обеспечивает высокую степень сжатия при

минимальной сложности установки, поскольку все необходимое для развертывания приложения содержится в единственном файле.

Однако в некоторых случаях требуется упаковать данные приложения в несколько CAB-файлов. При этом можно указать размер выходных CAB-файлов, что удобно, если размер выходных файлов ограничен. Например, если предполагается распространять приложение на дискетах, логично задать для CAB-файлов размер в 1440 кб (об этом — в следующем разделе). Полученные в результате CAB-файлы можно затем скопировать на дискеты.

Наконец, параметр Package Files позволяет скомпоновать выходные данные в виде набора несжатых файлов. При этом выходные файлы проекта просто копируются в тот же каталог, что и MSI-файл.

CAB Size

При выборе упаковки данных приложения в CAB-файлы становится доступным параметр, определяющий размер CAB-файла. Если назначить ему значение Unlimited, создается единственный CAB-файл, в который упаковываются все выходные файлы проекта. Если же выбрано значение Custom, максимальный размер выходных CAB-файлов задает пользователь — это удобно, если предполагается распространять приложение на дискетах или иных сменных носителях.

Bootstrapper

Этот параметр определяет, нужно ли при компиляции проекта генерировать программу, подготавливающую клиентский компьютер к развертыванию приложения. Перед установкой собственно приложения эта программа устанавливает на целевой компьютер пакет Windows Installer 1.5 — стандартную версию установочной программы Windows XP. Если выбран этот параметр, во время компиляции проекта установочной программы генерируются дополнительные файлы, которые при необходимости используются для установки Windows Installer 1.5 на клиентский компьютер. При развертывании на компьютерах под управлением Windows XP программа для загрузки Windows Installer не требуется, но, если предполагается развертывать приложение, требующее Windows Installer, на клиентах под управлением прежних версий Windows, без такой программы не обойтись, по крайней мере при первой установке.

Если планируется развертывать приложение на компьютерах под управлением Windows XP либо на компьютерах с предустановленным Windows Installer 1.5, параметру Bootstrapping можно назначить значение None, чтобы отказаться от генерации программы для установки Windows Installer 1.5. В противном случае следует выбрать значение Windows Installer Bootstrapper. В результате будут сгенерированы четыре дополнительных файла (таблица 10-1).

Если предполагается, что приложение будет загружаться из Интернета, выберите параметр Web Bootstrapper. Три дополнительных файла генерируются и записываются в заданный выходной каталог. Файлы InstMsiA.exe и InstMsiW.exe выполняют ту же функцию, что при выборе параметра Windows Installer Bootstrapper, — устанавливают программу Windows Installer на клиентский компьютер. Третий файл также называется Setup.exe, но его назначение несколько отличается: он определяет, установлен ли на целевом компьютере Windows Installer и, если нет, загружает соответствующую программу для его установки. После этого загружается и исполняется MSI-файл, который содержит собственно приложение.

Таблица 10-1. Файлы программы, устанавливающей Windows Installer

Файл	Описание
Setup.exe	Этот файл является точкой входа в процесс установки. При запуске программа Setup проверяет, установлен ли на данном компьютере Windows Installer 1.5 и, если нет, вызывает InstMsiA.exe или InstMsiW.exe, чтобы установить Windows Installer. После этого Windows Installer приступает к установке самого приложения, извлекая его из MSI-файла
InstMsiA.exe	Устанавливает Windows Installer 1.5 на компьютеры, работающие под управлением Windows 95/98
InstMsiW.exe	Устанавливает Windows Installer 1.5 на компьютеры, работающие пол. управлением Windows NT/2000
Setup.ini	Содержит имя MSI-файла, который Setup.exe обрабатывает после проверки наличия Windows installer

При выборе параметра Web Bootstrapper открывается диалоговое окно Web Bootstrapper Settings, в котором можно задать пару дополнительных свойств проекта. Свойство Setup folder URL указывает Web-каталог, в котором находится программа установки приложения со всеми связанными с ней файлами. Второе свойство, Windows Installer upgrade folder URL, определяет отдельный каталог для загружаемых через Web файлов, необходимых для установки Windows Installer (InstMsiA.exe и InstMsiW.exe). Таким образом, программы для установки Windows Installer и самого приложения можно хранить в разных Web-каталогах. Если указанные свойства не заданы, установочные файлы Windows Installer записываются в тот же каталог, что и установочные файлы приложения.

Compression

Эта параметр определяет схему сжатия для проекта установочной программы. Если сжатие не используется, этот параметр недоступен, в противном случае можно выбрать один из трех вариантов сжатия. Значение None отключает сжатие файлов. При этом установка приложения выполняется быстрее, но за это приходится расплачиваться существенно большим размером установочных файлов. Значение Optimized For Speed — компромисс между степенью сжатия файлов проекта и скоростью установки приложения, а Optimized For Size задает максимальное сжатие, что требует больше времени на установку.

Подпись Authenticode

Если вы хотите засвидетельствовать подлинность вашего кода, добавьте к проекту установочной программы цифровую подпись, Для этого пометьте флажок Authenticode Signature. Станут доступными три свойства, связанные с цифровой подписью, В поле Certificate File следует указать каталог с сертификатом подлинности, который будет использоваться для подписания файлов, — его можно найти, щелкнув кнопку Browse. В поле Private Key задают каталог с закрытым ключом, используемым для шифрования подписываемых файлов. Поиск каталога также осуществляется с помощью кнопки Browse. Наконец, в поле Timestamp Server URL указывают адрес сервера временных меток, используемый для подписания файлов. Первые два параметра (сертификат подлинности и закрытый ключ) являются обязательными

для создания цифровой подписи, подтверждающей подлинность кода, а третий (адрес сервера временных меток) — нет.

Компоновка, распространение и развертывание проекта

После добавления требуемых выходных файлов и определения всех необходимых свойств проекта установочной программы проект компонуют и приступают к распространению. После выполнения этих операций проект считается готовым к развертыванию на клиентских компьютерах.

Компоновка

Чтобы скомпоновать проект установочной программы, выделите его в окне Solution Explorer и выберите в меню Build одноименную команду — в результате начнется процесс компоновки проекта. Сгенерированные при компоновке файлы дистрибутива записываются в каталог, заданный в свойствах проекта (значение поля Output File Name), откуда их можно скопировать на выбранный для распространения носитель,

Распространение на сменных носителях

Приложения традиционно распространяют на сменных носителях. При выборе носителя следует учесть его емкость. Например, максимальная емкость дискеты составляет 1,44 Мб, что явно мало для большинства современных приложений. Компакт-диски обладают значительно большей емкостью, но для особо объемных проектов и ее может оказаться недостаточно. Если размер приложения превышает емкость носителя, выбранного для его распространения, можно упаковать выходные данные проекта в CAB-файлы и разместить полученный дистрибутив на нескольких носителях.

▶ **Распространение приложения на сменных носителях**

1. Оцените предполагаемый объем дистрибутива и задайте при помощи свойств приложения подходящий способ упаковки выходных файлов. Если вы планируете распространять свое приложение на дискетах, выберите упаковку в CAB-файлы и установите для них максимальный размер в 1440 кб.
2. Скомпилируйте проект установочной программы.
3. Скопируйте полученные файлы из каталога, куда они были записаны при компиляции, на подготовленные сменные носители. При использовании дискет для распространения приложения в виде набора CAB-файлов, скопируйте MSi-файл на первую дискету дистрибутива, а на остальные — CAB-файлы приложения.

Распространение через сеть

Если приложение предназначено для группы пользователей, подключенных к общей сети, дистрибутив удобно распространять через сеть. Это самый простой способ распространения приложений — для его реализации достаточно создать на сетевом сервере разделяемый каталог и скопировать туда установочные файлы приложения. После этого остается лишь опубликовать адрес каталога, чтобы все пользователи, заинтересованные в получении нового приложения, смогли самостоятельно переписать его на свои компьютеры. Администратор сети может защитить каталог с дистрибутивом приложения паролем, ограничив круг лиц, которым он доступен, целевой аудиторией приложения. Если приложение использует специальную

программу для подготовки к развертыванию, задайте для свойства Bootstrapper на страницах свойств проекта установочной программы значение Windows Installer Bootstrapper.

▶ **Распространение приложений через сеть**

1. Чтобы создать программу для подготовки целевых компьютеров к развертыванию, на страницах свойств проекта установочной программы задайте для свойства Bootstrapper значение Windows Installer Bootstrapper.
2. Скомпилируйте проект установочной программы.
3. Создайте на сетевом сервере разделяемый каталог и определите для него необходимые ограничения доступа.
4. Скопируйте в созданный каталог файлы, полученные при компиляции проекта установочной программы.
5. Сообщите сетевой адрес этого каталога предполагаемым пользователям приложения.

Применение административной установки

Для распространения приложений через сеть можно применять административную установку. Для реализации этого подхода в разделяемом каталоге создается образ приложения и установочная программа, после чего конечные пользователи смогут загрузить из этого каталога установочный файл и установить приложение на свои компьютеры. Такой способ обеспечивает больше гибкости в настройке парамегрон установки приложения на целевом компьютере. При использовании административной установки разрешается задавать определенную конфигурацию приложения, которая будет реплицирована на клиентские компьютеры.

▶ **Распространение приложения через сеть и с использованием административной установки**

1. Чтобы создать программу для подготовки целевых компьютеров к развертыванию приложения, на страницах свойств проекта установочной программы задайте для свойства Bootstrapper значение Windows Installer Bootstrapper.
2. Скомпилируйте проект установочной программы.
3. Создайте на сетевом сервере разделяемый каталог и определите для него необходимые ограничения доступа.
4. Вызовите окно командной строки.
5. В окне командной строки перейдите в каталог проекта установочной программы.
6. Чтобы инициировать установку, исполните следующую команду:

   ```
   msiexec.exe /A <setup>.msi
   ```

   где *<setup>* — имя установочной программы вашего приложения.
7. Следуйте указаниям мастера Setup Project, В качестве целевого каталога установки укажите разделяемый сетевой каталог, созданный ранее.
8. Сообщите сетевой адрес этого каталога предполагаемым пользователям приложения.

Распространение приложений через Интернет

Приложения, предназначенные для широкого крута пользователей, можно загружать на целевые компьютеры через Интернет. Процедуры подготовки приложения к распространению через Интернет и через локальную сеть не отличаются за ис-

ключением одного: файлы дистрибутива публикуют не в разделяемом каталоге, а в виртуальном Web-каталоге. Если для установки приложения требуется Windows Installer, скопируйте соответствующие файлы в отдельный каталог, заданный в поле Windows Installer upgrade folder URL на странице свойств проекта установочной программы. При этом необходимо установить свойство Bootstrapper в Web Bootstrapper, поскольку программы для установки Windows Installer, предназначенные для использования в Web, имеют встроенный механизм аутентификации.

▶ **Распространение приложений через Web**

1. Чтобы создать программу для подготовки целевых компьютеров к развертыванию приложения, на страницах свойств проекта установочной программы задайте для свойства Bootstrapper значение Web Bootstrapper. Запишите в свойство Setup Folder URL адрес виртуального Web-каталога, куда следует скопировать устанавливаемое приложение. При необходимости укажите адрес дополнительного каталога с программой установки Windows Installer при помощи свойства Windows Installer Upgrade Program URL.
2. Скомпилируйте проект установочной программы.
3. Скопируйте файлы, полученные при компиляции, в виртуальный каталог, предназначенный для публикации дистрибутива приложения. Если в пункте 1 вы задали адрес каталога с программой установки Windows Installer, скопируйте в этот каталог установочные файлы Windows Installer.
4. Опубликуйте URL приложения, чтобы он был доступен всем заинтересованным лицам.

Развертывание проекта

После создания дистрибутива приложения с установочной программой и распространения его по целевым компьютерам дальнейшее развертывание не представляет сложности. Если для установки приложения не требуется Windows Installer, достаточно дважды щелкнуть MSI-файл, сгенерированный при компиляции установочного проекта, в противном случае для запуска установки следует дважды щелкнуть файла Setup.exe, который при необходимости установит Windows Installer. В любом случае на следующем этапе открывается MSI-файл, запускающий мастер Setup — он автоматически устанавливает приложение в заданный каталог. О настройке мастера установки рассказано на занятии 2.

Резюме

- Для развертывания простых .NET-приложений применяют команду XCOPY, которая копирует на целевой компьютер каталог приложения со всем содержим. Это способ годится только в том случае, когда на всех целевых компьютерах установлена инфраструктура .NET Framework, а устанавливаемое приложение не зависит ни от каких внешних общих сборок или ресурсов. В силу указанных ограничений XCOPY сравнительно редко используют для развертывания приложений.
- В Visual Studio .NET можно создавать проекты установочных программ для приложений и дополнительных модулей, использующих Windows Installer. Для исполнения созданной таким образом установочной программы необходим Windows Installer 1.5. Установочный комплект (дистрибутив) приложения содержит все конфигурационные данные и все необходимое для установки приложения на целевой компьютер. Дополнительные модули применяют для распространения DLL-файлов, которые нельзя установить отдельно, а только вместе с приложением.
- Мастер Setup Project позволяет выбрать файлы для включения в проект установочной программы и тип проекта.
- Параметры компоновки, определяемые свойствами проекта установочной программы, позволяют задать способ упаковки файлов дистрибутива и степень сжатия, добавить программу для подготовки целевых компьютеров к развертыванию приложения, назначить каталог для записи дистрибутива, а также сконфигурировать цифровую подпись, подтверждающую подлинность кода,
- Предусмотрено несколько путей распространения дистрибутива, полученного компоновкой проекта установочной программы. Чтобы дистрибутив подходил для выбранного способа распространения приложения, следует соответствующим образом настроить параметры компоновки для проекта установочной программы.

Занятие 2. Конфигурирование проекта установочной программы

Проекты установочных программ, использующие Windows Installer, предоставляют богатые возможности настройки: разрешается указывать каталог для установки файлов на целевом компьютере, создавать связи между файлами; проверять условия, чтобы разрешить или запретить установку приложения, а также выполнять различные действия, определяемые разработчиком. Это занятие посвящается созданию и настройке установочных программ, использующих Windows Installer.

Изучив материал этого занятия, вы сможете:

- ✓ опубликовать сведения о приложении при помощи соответствующих свойств проекта установочной программы;
- ✓ настраивать структуру каталогов для развертывания приложения с помощью редактора файловой системы;
- ✓ создавать разделы реестра во время установки при помощи редактора реестра;
- ✓ создавать связи между файлами и приложениями с помощью редактора типов файлов;
- ✓ модифицировать пользовательский интерфейс установочной программы при помощи редактора пользовательского интерфейса;
- ✓ определять нестандартные действия, исполняемые во время установки приложения;
- ✓ проверять условия, необходимые для установки приложения;
- ✓ создавать с помощью программы Ngen.exe образ приложения, скомпилированного в машинные коды, предназначенный для клиентских компьютеров;
- ✓ проверять соответствие разрешений, запрошенных устанавливаемым приложением, политике безопасности.

Продолжительность занятия - около 45 минут.

В большинстве случаев для создания, компиляции и распространения дистрибутива приложения достаточно сведений из занятия 1. Тем не менее иногда проект установочной программы приходится настраивать дополнительно. В Visual Studio .NET предусмотрены инструменты, позволяющие задавать различные свойства установочной программы и даже определять нестандартные действия, которые она будет выполнять в ходе установки приложения. После установки приложения при помощи специальной утилиты командной строки можно создать прекомпилированный образ приложения и поместить его в кэш образов, а также проверять политики безопасности для установленного приложения.

Проект установочной программы

Проект установочной программы поддерживает ряд свойств, которые позволяют предоставить некоторые сведения о проекте, а также задать определенные действия, что удобно для управления версиями проекта. Их устанавливают в окне Properties (здесь имеется в виду окно Properties интегрированной среды разработки, а не стра-

ницы свойств проекта, о которых шла речь на предыдущем занятии). Большинство из них касается сведений, описывающих приложения:

- **AddRemoveProgramsIcon** — задает значок, отображаемый в диалоговом окне Add/Remove Programs на клиентском компьютере;
- **Author** — сведения об авторе программы;
- **Description** — описание приложения;
- **Keywords** — список ключевых слов, связанных с приложением;
- **Localization** — сведения о локализации приложения;
- **Manufacturer** — информация о производителе приложения;
- **ManufacturerURL** — URL Web-сайта производителя;
- **ProductName** — имя продукта;
- **Subject** — название области применения приложения;
- **SupportPhone** — номер телефона службы технической поддержки приложения;
- **SupportURL** — адрес сайта службы поддержки технической поддержки приложения;
- **Title** — название приложения.

Прочие свойства проекта установочной программы определяют ее поведение во время установки приложения:

- **DetectNewerInstall** — если установить это свойство в *true,* программа установки попытается найти на целевом компьютере более «свежую» версию данного приложения. Если таковая отыщется, установка приложения прервется;
- **RemovePreviousVersion** — если установить это свойство в *true,* программа установки попытается найти на целевом компьютере предыдущие версии данного приложения и удалит их, если таковые отыщутся;
- **Version** — хранит сведения, используемые предыдущими свойствами для определения версии приложения.

Кроме того, поддерживаются два дополнительных свойства: ProductCode и UpgradeCode — ни в коем случае не изменяйте их вручную.

Значения всех указанных свойств задают во время разработки в окне Properties. При необходимости это можно сделать и для отдельных файлов проекта установочной программы.

Регистрация компонентов во время установки

Если проект установочной программы включает компоненты Font, COM или любые другие компоненты, требующие регистрации, во время установки это необходимо сделать. Для этого следует установить в окне Properties свойство *Register* для соответствующего файла. Оно принимает следующие значения:

- **vsdrpDoNotRegister** — объект не требует регистрации;
- **vsdrpCOM** — объект будет зарегистрирован как объект COM;
- **vsdrpCOMRelativePath** — объект будет зарегистрирован как изолированный объект COM;
- **vsdrpCOMSelfReg** — во время установки этот элемент регистрируется как объект COM, регистрирующийся самостоятельно;
- **vsdrpFont** — во время установки этот объект будет зарегистрирован как компонент Font.

Для .NET-сборок и компонентов регистрация не требуется, поэтому для них данное свойство можно установить в *vsdrpDoNotRegister* для шрифта — в *vsdrpFont*, а для файлов COM, содержащих в сведениях о версии строку OleSelfRegister, — в vsdrpCOMSelfReg.

▶ **Как зарегистрировать компонент во время установки**

Выделите этот нужный компонент в окне Solution Explorer. В окне Properties установите для его свойства *Register* соответствующее значение.

Редакторы **свойств программы установки**

В Visual Studio .NET имеется шесть редакторов свойств установочной программы, позволяющие настраивать самые разные ее свойства:

- **редактор файловой системы (File System Editor)** — позволяет выбрать каталог файловой системы целевого компьютера, в который следует установить приложение;
- **редактор реестра (Registry Editor)** — создает записи реестра в процессе установки;
- **редактор типов файлов (File Types Editor)** — связывает типы файлов с обрабатывающими их приложениями;
- **редактор пользовательского интерфейса (User Interface Editor)** — модифицирует пользовательский интерфейс установочной программы. Используется как при обычной установке, так при установке с помощью сценария;
- **редактор нестандартных действий (Custom Actions Editor)** — задает нестандартные действия, выполняемые установочной программой;
- **редактор условий (Launch Conditions Editor)** — определяет необходимые условия для начала установки.

Далее эти редакторы обсуждаются более подробно.

Редактор файловой системы

Этот редактор позволяет манипулировать файловой системой целевого компьютера: записывать выходные файлы в различные каталоги, создавать на целевом компьютере новые каталоги, а также создавать или добавлять ярлыки. Чтобы запустить редактор файловой системы, выделите свой проект в окне Solution Explorer и выберите в меню View команду Editor\File System Editor.

Как видно из рис. 10-8, окно редактора разделено на две панели.

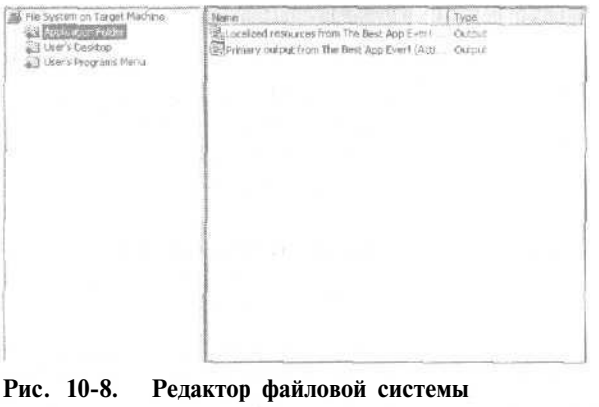


Рис. 10-8. Редактор файловой системы

На правой панели отображается список выходных файлов проекта установочной программы, а на левой — структура каталогов целевого компьютера. Исходно она включает три папки: каталог приложения, рабочий стол и папку Программы для текущего пользователя. По умолчанию выходные файлы записываются в каталог приложения. Целевой каталог для любого файла можно изменить, выделив этот файл на правой панели и перетащив его в требуемую папку. Кроме того, редактор файловой системы позволяет создавать дополнительные папки — для этого достаточно щелкнуть его левую панель правой кнопкой и выбрать из контекстного меню команду Add Special Folder (рис, 10-9).

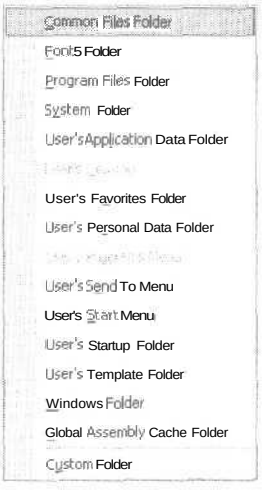


Рис. 10-9. Контекстное меню Special Folders

Здесь можно добавить к структуре каталогов стандартную папку либо создать собственную. Папки, созданные разработчиком при помощи редактора, создаются в файловой системе целевого компьютера во время установки приложения.

Добавление несжатого файла к дистрибутиву

Редактор файловой системы позволяет добавлять к проекту установочной программы несжатые файлы и сборки. Они включаются в дистрибутив и устанавливаются одновременно с приложением, но не сжимаются и не упаковываются с другими выходными файлами проекта установочной программы.

▶ **Добавление к дистрибутиву несжатых файлов и сборок**

1. Если целевой каталог для несжатого файла отсутствует на левой панели редактора, щелкните ее правой кнопкой и добавьте нужный каталог с помощью контекстного меню. Так, в любом из существующих каталогов разрешается создать вложенный каталог, щелкнув существующий каталог правой кнопкой и выбрав команду Add\Folder.
2. На левой панели редактора файловой системы щелкните правой кнопкой каталог, в который следует установить несжатый файл, и выберите File (или Assembly, если добавляется сборка).
3. Выберите нужный файл и щелкните OK (в случае сборки) или Open (в случае файла) — файл (или сборка) добавляется к проекту установочной программы; во время установки приложения этот файл будет записан в соответствующий целевой каталог на целевом компьютере.

Размещение сборок в GAC во время установки приложения

Среди прочего редактор файловой системы позволяет помещать сборки в GAC во время установки приложения. Для этого сборки добавляют к проекту установочной программы в виде несжатых файлов. Соответствующие файлы добавляются к проекту, но не сжимаются и не упаковываются с остальными выходными файлами. Чтобы установить сборку в GAC, ее прежде всего необходимо подписать при помощи строгого имени (подробнее о строгих именах — в главе 9). После этого останется лишь добавить каталог GAC к структуре каталогов в редакторе файловой системы и поместить нужную сборку в этот каталог — во время установки приложения сборка будет записана в GAC.

▶ **Как поместить сборку в GAC во время установки приложения**

1. Подпишите сборку при помощи строгого имени (как это делается — см. в главе 9).
2. Создайте проект установочной программы и откройте редактор файловой системы.
3. Щелкните правой кнопкой левую панель редактора и выберите Add Special Folder, затем выберите в контекстном меню каталог GAC.
4. Щелкните правой кнопкой каталог GAC на левой панели редактора файловой системы и выберите команду Add\Assembly. Найдите требуемую сборку и щелкните OK, чтобы добавить ее. Если эта сборка до сих пор не добавлена к проекту установочной программы, это происходит сейчас. Во время установки приложения она помещается в GAC.

Добавление ярлыка при помощи редактора файловой системы

При желании для устанавливаемого приложения можно создать ярлык на рабочем столе (или в другой папке) целевого компьютера. Для этого также применяют редактор файловой системы.

▶ **Как создать ярлык на целевом компьютере**

1. В окне Solution Explorer щелкните правой кнопкой файл, для которого необходимо создать ярлык, и выберите Choose Find In Editor — на правой панели редактора файловой системы отображается каталог с выбранным файлом.
2. На правой панели редактора файловой системы щелкните правой кнопкой файл, для которого создаете ярлык, и выберите команду Create Shortcut — ярлык для этого файла создается и добавляется на панель.
3. Перетащите ярлык с правой панели в нужный каталог, отображаемый на левой панели.

Редактор реестра

Редактор реестра (рис. 10-10) позволяет во время установки приложения создавать записи в системном реестре. Разрешается создавать новые разделы реестра и устанавливать новые значения для существующих разделов.

Окно редактора реестра также разделено на две панели; на левой панели отображается реестр целевого компьютера, а на правой — значения текущего раздела.

▶ **Как добавить к реестру новый раздел**

Чтобы записать в реестр новый раздел, щелкните правой кнопкой существующий раздел, в котором нужно создать вложенный раздел, и выберите команду New Key — создается новый вложенный раздел.

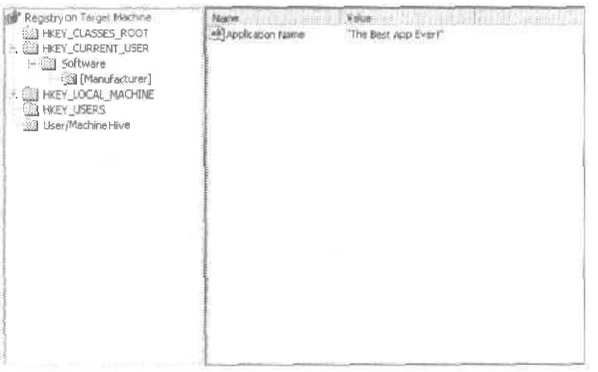


Рис. 10-10. Редактор реестра

▶ **Как добавить новое значение в существующий раздел реестра**

1. На левой панели щелкните правой кнопкой раздел реестра, к которому вы хотите добавить новое значение, и выберите команду New, затем, в зависимости от типа добавляемого значения, щелкните String Value, Environment, String Value. Binary Value или DWORD Value.
2. В окне Properties определите свойство *Value* добавленного значения реестра,

Редактор типов файлов

Этот редактор (рис. 10-11) позволяет создавать связать устанавливаемое приложение с определенными файловыми расширениями. Пример — связь между файлами с расширением .txt и Блокнотом: двойным щелчком любой .txt-файл автоматически открывается в Блокноте.

Рис. 10-11. Редактор типов файлов

Чтобы создать новую связь «приложение — тип файлов», щелкните правой кнопкой File Types Editor и выберите Add File Type — в окне появится новая запись для типа файла, ниже отображается действие *&Open*. Чтобы добавить дополнительные действия, щелкните правой кнопкой созданную запись и выберите Add Action. Настроить созданную связь можно через окно, определяющее свойства типа файлов и заданного для него действия.

Так, свойство связи *Name* определяет составляющие ее имена типов файлов. Чтобы добавить к связи дополнительные типы файлов, задайте соответствующие файловые расширения в свойстве *Extensions*. Если с устанавливаемым приложением нужно связать несколько типов файлов, перечислите их в данном свойстве через точку с запятой (точка перед расширением не обязательна).

Свойство *Command* связи представляет приложение, запускаемое в ответ на выполнение некоторого действия над файлами, заданными свойством *Name*. Так, если нужно связать устанавливаемое приложение и файлы с расширением .tew, поместите путь к приложению в свойство *Command* — при исполнении некоторого действия над .tew-файлом (например, при двойном щелчке) заданное приложение запустится и обработает файл.

У связанного с типом файла действия есть три свойства. Свойство *Name* содержит текст, который отображается в контекстном меню, вызываемом щелчком файла с данным расширением правой кнопкой. Свойство *Verb* задает описание действия, выполняемого над данным типом файлов, а свойство *Arguments* позволяет передать приложению любые необходимые ему данные через аргументы командной строки.

Если щелкнуть правой кнопкой файл документа, связанный с некоторым приложением, открывается контекстное меню, отображающее значения свойства *Name* для всех действий, определенных для данного типа файлов. При выборе одного из этих действий запускается заданное свойством *Command* приложение, которое получает через аргументы командной строки путь к данному документу, а также любые параметры, заданные свойством *Arguments*. Путь к файлу всегда является первым элементом массива аргументов. Все аргументы передаются методу *Main* вызываемого приложения. Естественно, в приложении должен быть код, обрабатывающий переданные аргументы и исполняющий заданное действие. Следующий пример демонстрирует, как при помощи команды *Open* приложения открыть файл, заданный аргументами командной строки:

Visual Basic .NET

```
Shared Sub Main(ByVal args() As String)
   If args.Length <> 0 Then
      ' Предполагается, что метод Open принимает в качестве параметра
      ' строку, содержащую путь к открываемому файлу,
      ' Метод Open должен быть статическим, поскольку из
      ' таких методов нельзя вызывать методы экземпляров.
      Open(args(0))
   Else
      Application.Run(New Form1())
   End If
End Sub
```

Visual C#

```
static void Main(string[] args)
{
   if(args.Length!=0)
   {
      // Предполагается, что метод Open принимает в качестве параметра
      // строку, содержащую путь к открываемому файлу.
      // Метод Open должен быть статическим, поскольку из
      // таких методов нельзя вызывать методы экземпляров.
      Open(args[0]);
   }
```

```
    else
    {
        Application.Run(new Form1());
    }
}
```

Примечание У приложений Windows Forms, написанных на Visual Basic, по умолчанию нет процедуры *Main*, поэтому ее придется добавить самостоятельно и сделать *Sub Main* стартовым объектом, установив соответствующее свойство проекта. В приложениях на Visual C# необходимо вручную откорректировать объявление метода *static void Main,* чтобы заставить его принимать в качестве аргумента массив строк.

Редактор пользовательского интерфейса

Этот редактор позволяет модифицировать пользовательский интерфейс установочной программы приложения. Редактор пользовательского интерфейса отображает два дерева объектов: одно для обычной установки, а другое для установки администратором (рис. 10-12).

Рис. 10-12. Редактор пользовательского интерфейса

Оба дерева в графической форме представляют набор диалоговых окон, отображаемых пользователю во время установки приложения. У каждого дерева три ветви: Start (начало), Progress, (продолжение) и End (завершение), содержащих набор предопределенных диалоговых окон, которые отображаются на соответствующем этапе установки.

В начальной фазе установки программа Windows Installer собирает сведения о компьютере и пользователе, необходимые для развертывания приложения. Все диалоговые окна, требующие взаимодействия с пользователем (например окно выбора каталога для установки), следует помещать в ветвь начальной фазы. На заключи-

тельном этапе начальной фазы Windows Installer проверяет размер доступного места на диске. Если его не хватает, установка заканчивается неудачей.

Во время второй фазы (продолжения) выполняется собственно установка приложения (копируются его файлы). На этой фазе взаимодействие с пользователем обычно не требуется, отображается только окно с индикатором хода установки.

Заключительная фаза установки начинается после завершения копирования файлов приложения. В течение этой фазы пользователь также получает сведения о ходе установки.

Разрешается редактировать свойства любого диалогового окна — для этого выделите нужное окно в редакторе пользовательского интерфейса и установите его свойства в окне Properties. Например, изменив свойство *BannerBitmap* диалогового окна Welcome, можно заставить его отображать логотип своей компании,

Также можно изменить порядок отображения диалоговых окон в ходе установки приложения. Для этого щелкните диалоговое окно и перетащите его в требуемую точку дерева. Технически порядок отображения окон можно изменять произвольно, но на практике существует ряд ограничений, определяемых здравым смыслом; например, нельзя начинать установку с отображения окна Finished.

Обычная и административная установка

При конфигурировании проекта для использования административной установки (см. занятие 1), можно предоставить администратору возможности, недоступные при обычной установке. К примеру, лишить конечных пользователей права выбора каталога для установки приложения, предоставив его администратору, курирующему установку. В этом случае из набора окон для обычной установки следует удалить окно Installation Folder, но в наборе окон для установки администратором его следует оставить. Так, если администратор задаст каталог для установки приложения, конечные пользователи смогут установить это приложение только в него.

Добавление диалоговых окон

К пользовательскому интерфейсу программы установки можно добавить дополнительные диалоговые окна, настраивая их по своему вкусу. Например, можно добавлять самые разные диалоговые окна, способные принимать информацию от пользователя через переключатели, флажки и текстовые поля. Внешний вид этих диалоговых окон разрешается настраивать, редактируя соответствующие свойства в окне Properties. Помимо свойств, определяющих внешний вид окна (таких, как *BannerBitmap*, *BannerText* и *BodyText*), поддерживается ряд свойств, позволяющих передать приложению информацию, введенную пользователем. Рассмотрим диалоговое окно типа RadioButtons (окно с двумя переключателями). Можно добавить это окно к набору окон программы установки. Для этого в окне редактора интерфейса щелкните соответствующий узел правой кнопкой и выберите команду Add Dialog — откроется одноименное диалоговое окно; укажите в нем окно типа RadioButtons (2 Buttons). Помимо стандартных свойств, определяющих внешний вид, это окно поддерживает следующие свойства:

- *Button 1 Label*
- *Button2Label*
- *Button1Value*
- *Button2Value*
- *DefaultValue*
- *ButtonProperty*

Свойства вида *Button*N*Label*, где N — номер кнопки переключателя, определяют текст ее подписи, а свойства вида *Button*N*Value* — значение, возвращаемое группой переключателей при выборе переключателя с номером N. Свойство *DefaultValue* представляет первоначальное значение группы переключателей, а в свойстве *ButtonProperty* находится строка с именем свойства — его программа установки использует для получения значения этой группы переключателей. Это свойство вызывают при проверке нестандартных условий во время установки (о проверке условий см. ниже), чтобы решить, продолжать ли установку. Так, если *ButtonProperty*= *Buttons,* можно определить условие, которое во время установки будет проверять значение свойства *Buttons.* Если выбран первый переключатель, *Buttons* получит значение, определяемое свойством *Button1Value*, а если выбран второй переключатель — значение, заданное свойством *Button2Value.* Подобные свойства поддерживают многие диалоговые окна, добавляемые к программе установки, — это позволяет при установке приложения предоставить пользователям самые разные возможности вплоть до самых сложных.

Редактор нестандартных действий

Технология *нестандартных действий* (custom actions) обеспечивает поддержку дополнительных возможностей для программ установки. Редактор нестандартных действий позволяет добавить код, выполняемый в процессе установки. Этот код вызывается в ответ на одно из четырех событий, генерируемых программой установки, - *Install, Commit, Rollback* или *Uninstall.* Событие *Install* генерируется после копирования файлов, но перед подтверждением завершения установки. Событие *Commit* генерируется после подтверждения завершения установки на целевом компьютере. Действия, заданные для события *Rollback*, выполняются в случае неудачи установки и включают отмену (откат) изменений, внесенных программой установки. Наконец событие *Unmstall* генерируется при выборе удаления приложения. Редактор нестандартных действий (рис. 10-13) позволяет привязать код к этим событиям.

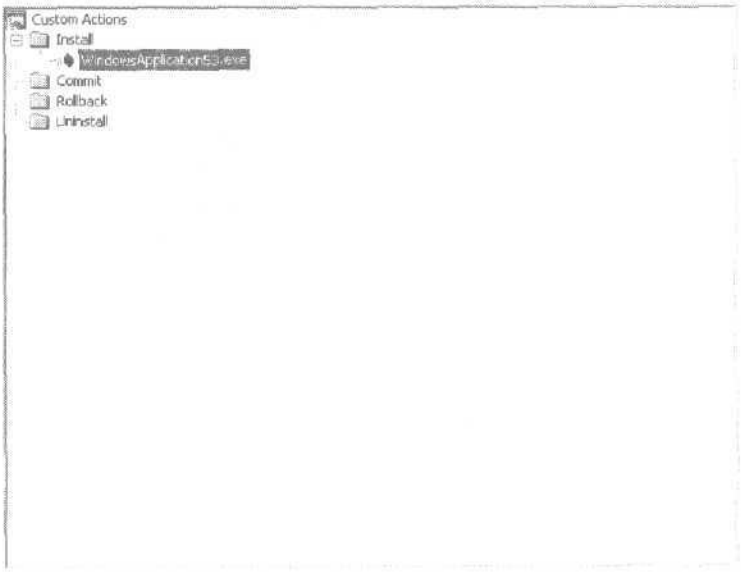


Рис. 10-13. Редактор нестандартных действий

Любой код может быть исполнен как нестандартное действие. Чтобы создать нестандартное действие, в окне редактора щелкните правой кнопкой требуемое событие и выберите из контекстного меню команду Add Custom Action — откроется диалоговое окно Select Item In Project. В этом окне можно назначить действию какой-либо элемент проекта — в результате созданное нестандартное действие, представляющее заданный элемент проекта, добавляется к проекту установочной программы. Свойства нестандартного действия (таблица 10-2) настраивают через окно Properties.

Таблица 10-2. Свойства нестандартного действия

Имя	Описание
(Name)	Отображаемое имя для данного нестандартного действия
Arguments	Передает параметры командной строки программе, реализующей нестандартное действие. Годится только для нестандартных действий, реализованных исполняемыми программами (.exe-файлами)
Condition	Позволяет задать логический оператор, вычисляемый перед выполнением нестандартного действия. Если оператор дает *true,* действие выполняется, в противном случае — нет. Заданный этим свойством оператор также может обрабатывать значения, полученные от пользователя через дополнительные диалоговые окна
CustomActionData	Передает программе, реализующей нестандартное действие, дополнительные данные
Entry Point	Определяет имя метода нестандартного действия. Если это свойство не задано, вызывается метод, имя которого совпадает с именем события, срабатывание которого инициировало данное нестандартное действие (например, *Install*). Применимо только к нестандартным действиям, реализованным посредством DLL
InstallerClass	Хранит значение типа *Boolean,* указывающее, реализовано ли данное нестандартное действие в классе *Installer.* Если это так, его значение равно *true,* в противном случае — *false*
SourcePath	Содержит полный путь (на компьютере разработчика) к файлу программы, реализующей нестандартное действие; доступно только для чтения

▶ **Создание нестандартного действия**

1. Напишите, протестируйте и отладьте код, реализующий нестандартное действие, которое вы хотите добавить к проекту установочной программы.

   Затем в меню View выберите Editors\Custom Actions Editor, чтобы вызвать редактор нестандартных действий.
2. Щелкните правой кнопкой событие, с которым следует связать нестандартное действие, и выберите команду Add Custom Action — откроется окно Select Item In Project.
3. Найдите и выберите созданный в пункте 1 файл с кодом, реализующий нестандартное действие.
4. В окне Properties настройте свойства для нового нестандартного действия.

Редактор условий установки

Редактор условий установки — Launch Conditions Editor (рис. 10-14) — позволяет определять условия, которым целевой компьютер должен удовлетворять для начала установки. Так, можно требовать, чтобы целевой компьютер работал под определенной версией Windows или чтобы на нем была установлена определенная версия некоторого компонента, необходимого приложению. Кроме того, можно проверять наличие определенных файлов, записей реестра, компонентов Windows Installer и принимать решение о начале установки по результатам проверки.

Окно редактора разделено на две части; в первой задают объект (файл, раздел реестра или компонент Windows Installer), а во второй — условие, зависящее от наличия на целевом компьютере этого объекта. Если условие выполняется, процесс установки продолжается, в противном случае установка прекращается с откатом действий, сделанных к данному моменту,

Рис. 10-14. Редактор условий установки

Выбор объекта условия

Чтобы задать объект условия, в редакторе условий щелкните правой кнопкой Search Target Machine и выберите требуемый объект (таблица 10-3).

Таблица 10-3. Типы объектов условий установки

Тип	Объект на целевом компьютере
File Search	Файл с заданной версией, максимальным или минимальным размером или датой
Registry Search	Запись в реестре (корень, раздел и значение)
Windows Installer Search	Компонент Windows Installer

Каждый объект условия поддерживает свойство *Property*, которое задает имя искомого объекта, — оно используется для получения булева значения, представляющего результат поиска. Имя, заданное этим свойством, используется для конструирования условий установки (об этом далее в этом разделе).

▶ **Как задать объект условия установки**

1. В меню View выберите Editor\Launch Conditions, чтобы вызвать редактор условий установки.
2. В редакторе условий щелкните правой кнопкой Search Target Machine и выберите в контекстном меню требуемый объект.
3. В окне Properties настройте условия поиска выбранного объекта.

Конструирование условий установки

Определив объект, можно проверить его наличие на целевом компьютере и сконструировать условие установки, зависящее от результатов этой проверки. Условие установки действует по принципу «все или ничего» и определяет дальнейшую судьбу установки приложения. Если это условие выполняется, установка продолжается, в противном случае она прекращается.

Чтобы создать условие установки, в редакторе условий щелкните правой кнопкой Launch Conditions и выберите команду Add Launch Condition. Свойства условий установки приводятся в таблице 10-4.

Таблица 10-4. Свойства условия установки

Имя	Что определяет
Name	**Имя условия** установки, **отображаемое в редакторе**
Condition	**Строку** с допустимым **логическим оператором, вычисляемым во время установки. Если оператор дает** *true,* установка **продолжается, а** если ***false*** — прекращается. В строке может **быть** несколько **выражений,** связанных **операторами AND, OR и др.**
Message	Текст **сообщения, отображаемого при неудачном завершении установки**

► **Как создать условие установки**

1. В меню View выберите команду Editor\Launch Conditions Editor, чтобы вызвать редактор условий установки.
2. В редакторе условий щелкните правой кнопкой Launch Conditions и выберите команду Add Launch Condition, чтобы добавить новое условие установки.
3. В окне Properties настройте свойства созданного условия.

Установка сборок в виде компилированных образов

Приложения и сборки, созданные с применением .NET Framework, обычно развертывают в виде файлов с кодом на языке IL. Во время выполнения этот IL-код компилируются по требованию в машинные коды, специфичные для данного компьютера, что обеспечивает максимально эффективное использование ресурсов и высокую производительность приложения. Если производительность имеет решающее значение, можно заранее скомпилировать приложение в машинные коды. Для создания образа приложения, заранее скомпилированного в машинные коды, предназначена утилита Ngen.exe.

Ngen.exe — это утилита командной строки, которая генерирует образ приложения или сборки, состоящий из машинных кодов, и помещает его в кэш образов (Native Image Cache). После этого при каждом вызове данной сборки CLR будет автоматически обнаруживать и использовать ее образ.

► **Создание образа сборки или приложения, компилированного в машинные коды**

1. Вызовите окно командной строки Visual Studio .NET. Для этого в меню Пуск выберите Программы\Microsoft Visual Studio .NET\Microsoft Visual Studio .NET Tools.
2. Перейдите в каталог с нужным приложением или сборкой.
3. Воспользуйтесь утилитой Ngen.exe, как показано ниже. Для компиляции библиотечных сборок достаточно указать имя сборки, а для исполняемых сборок требуется путь к .exe-файлу:

```
Ngen.exe myAssembly
Ngen.exe C:\myApp.exe
```

Проверка разрешений, запрашиваемых установленной сборкой

Установив сборку, можно воспользоваться утилитой Permview.exe для просмотра разрешений, запрашиваемых данной сборкой. Чтобы вызвать эту утилиту, откройте окно командной строки Visual Studio .NET и исполните одну из следующих команд:

```
Permview.exe myAssembly.dll
Permview.exe myApp.exe
```

где myAssembly — имя библиотечной сборки, а myApp — имя приложения. Предполагается, что декларация сборки хранится вместе со сборкой, в противном случае при вызове Permview.exe следует указать файл с декларацией сборки,

Эта утилита отображает список разрешений, запрошенных на уровне сборки. Если необходимо просмотреть полный список разрешений, включая запрошенные на уровне класса или метода, вызовите утилиту с параметром /DECL:

```
Permview.exe /DECL myAssembly.dll
```

Выходные данные этой утилиты можно направлять не только на консоль, но и в файл, который задают параметром /OUTPUT:

```
Permview.exe /OUTPUT myTextFile.text myAssembly.dll
```

Резюме

- Установочные программы обладают широкими возможностями настройки. Во время разработки через окно Properties разрешается задать свойства установочной программы, определяющие ее поведение и предоставляющие сведения о ее происхождении.
- Свойство *Register* предназначено для регистрации компонентов COM и шрифтов во время установки приложений.
- Редактор файловой системы позволяет модифицировать файловую систему целевого компьютера.
- Редактор реестра служит для добавления записей в реестр целевого компьютера.
- Редактор типов файлов предназначен для связывания устанавливаемого приложения с определенными типами файлов на целевом компьютере.
- Редактор пользовательского интерфейса позволяет настраивать интерфейс установочной программы.
- Редактор нестандартных действий позволяет определить нестандартные действия, выполняемые в процессе установки приложения.
- Редактор условий установки применяют для проверки наличия на целевом компьютере необходимых объектов и конструирования условий, разрешающих продолжить установку или прерывающих ее.
- Утилита Ngen.exe позволяет создать образ приложения или сборки, компилированной в машинные коды.
- Утилита Permview.exe предназначена для просмотра требуемых сборкой разрешений.

Практикум 10. Создание установочной программы

Вы создадите проект установочный программы для приложения, созданного на практикумах глав 8 и 9. Для решения поставленной задачи вы воспользуетесь мастером Setup Project и настроите полученную заготовку проекта при помощи страницы его свойств. Вы также должны добавить на рабочий стол целевого компьютера ярлык для приложения, а также при помощи условия установки проверить наличие ОС Windows 2000 или выше. В завершение вы скомпонуете созданный проект и установите приложение на свой компьютер, а затем проверите разрешения, необходимые вашему приложению. Решение задач этого практикума можно найти в каталоге \Labs\ChlO\SoIution на прилагаемом компакт-диске.

Прежде всего

Для выполнения задач этого практикума вам понадобятся результаты практикумов глав 8 и 9; можно также воспользоваться готовым решением, загрузив его из каталога \Labs\Ch10\Partial с прилагаемого компакт-диска.

Продолжительность практикума - около 30 минут.

Упражнение 1. Создание проекта установочной программы

Сейчас вы создадите проект установочной программы для своего приложения с помощью мастера Setup Project.

▶ **Создание проекта установочной программы**

1. В меню File выберите команду Add Project\New Project — откроется окно Add New Project.
2. На левой панели этого окна выберите Setup And Deployment Projects, а на правой — Setup Wizard. Назовите проект **International**Sales и щелкните OK.
3. В окне мастера Setup щелкните кнопку Next, чтобы перейти к странице выбора типа проекта.
4. Выберите Create A Setup For A Windows Application и щелкните Next, чтобы перейти к странице выбора файлов, включаемых в проект.
5. Выберите Primary Output, Localized Output и Content Files. Щелкните Next.
6. На четвертой странице щелкните Next, затем Finish, чтобы создать установочный проект.
7. В окне Solution Explorer щелкните правой кнопкой InternationalSales и выберите Properties — откроется окно свойств проекта InternationalSales.
8. В раскрывающемся меню Bootstrapper выберите None, чтобы отказаться от создания программы, устанавливающей Windows Installer.

Примечание Если ваш компьютер работает под управлением Windows 2000, вместо None нужно выбрать Windows Installer Bootstrapper.

9. Щелкните OK, чтобы закрыть окно свойств проекта.
10. Сохраните результаты своей работы, щелкнув File\Save All.

Упражнение 2. Настройка установочной программы

Сейчас вы создадите для устанавливаемого приложения ярлык, размещаемый на рабочем столе целевого компьютера, и добавите условие установки, проверяющее наличие на целевом компьютере Windows 2000 или более высокой версии ОС.

▶ **Добавление ярлыка для приложения**

1. В меню View выберите Editors\File System, чтобы вызвать редактор файловой системы.
2. На левой панели редактора щелкните Application Folder, чтобы открыть каталог со своим приложением — на правой панели редактора отображается список выходных файлов проекта установочной программы,
3. На правой панели редактора щелкните правой кнопкой Primary Output и выберите команду Create Shortcut — будет создан новый ярлык. Назовите его **International Sales.**
4. Перетащите ярлык в папку рабочего стола пользовательского компьютера.
5. В меню View выберите Editors\Launch Conditions Editor, чтобы вызвать редактор условий установки.
6. Щелкните правой кнопкой Launch Conditions и выберите команду Add Launch Condition, чтобы добавить новое условие установки. Назовите созданное условие **Windows2000**.
7. В окне Properties установите свойство Condition в **VersionNT>=500** — это потребует наличия на целевом компьютере ОС Windows 2000 или более поздней версии. Присвойте свойству Message значение **You must be running Windows 2000 or later to install this application.**
8. В меню Build выберите команду Build Solution, чтобы скомпилировать решение.
9. В меню File выберите команду Save all.
10. Закройте Visual Studio .NET.

Упражнение 3. Установка приложения

Сейчас вы установите приложение на свой компьютер, а также проверите работу ярлыка, созданного установочной программой, и политику безопасности вашей сборки.

▶ **Как установить приложение**

1. Вызовите Проводник и найдите установочную программу в каталоге Debug, вложенном в каталог Setup Project.
2. Дважды щелкните значок InternationalSales. Следуйте указаниям программы установки,
3. После завершения установки найдите на рабочем столе ярлык International Sales. Дважды щелкните его, чтобы запустить свое приложение.
4. Закройте приложение.
5. Вызовите командную строку Visual Studio .NET. С помощью команды cd перейдите в каталог, куда установлено ваше приложение.

6. С помощью утилиты Permview.exe просмотрите все разрешения, требуемые вашей сборкой. Чтобы отобразить полный список, вызовите Permview.exe с параметром /DECL:

```
Permview.exe /DECL Chapter10.exe
```

Примечание Если имя сборки, передаваемое утилите Permview.exe через параметр командной строки, содержит пробелы, заключите его в кавычки.

Закрепление материала

Приведенные ниже вопросы помогут вам лучше усвоить основные темы данной главы. Если вы не сумеете ответить на вопрос, повторите материал соответствующего занятия. Ответы для самопроверки — в приложении «Вопросы и ответы» в конце книги.

1. Как выполнить развертывание приложения с помощью XCOPY? В каких случаях его можно применять, а в каких — нет?

2. Согласно желанию заказчика приложение должно устанавливаться на компьютеры персонала через сеть, при этом для всех сотрудников его следует установить в один и тот же каталог. Как решить эту задачу?

3. Документация на ваше приложение создана в виде набора HTML-файлов. Как включить ее в состав дистрибутива приложения? Назовите минимум два способа.

4. Что такое образ приложения, скомпилированного в машинные коды? Как его создать?

5. Что делает программа, подготавливающая целевой компьютер к развертыванию приложения (bootstrapper)? В каких случаях она необходима?

ПРИЛОЖЕНИЕ

Вопросы и ответы

Глава 1. Введение в .NET Framework

1. Кратко перечислите главные компоненты .NET Framework и опишите, для чего они предназначены.

   .NET Framework состоит из двух основных частей: общеязыковой исполняющей среды (CLR), которая контролирует исполнение приложений, осуществляет проверку типов и управляет памятью, и библиотеки базовых классов .NET, содержащей тысячи готовых классов для разработки приложений.

2. Дайте определение ссылочным типам и типами значения.

   Тип значения содержит реальное значение переменной, которую он представляет, тогда как ссылочный тип вместо значения переменной содержит ссылку на область памяти, в которой оно реально находится.

3. Как ссылаться на члены библиотеки базовых классов .NET при помощи их сокращенных имен?

   Для ссылки на пространства имен .NET Framework применяют ключевые слова *Imports* (Visual Basic .NET) и *using* (Visual C#).

4. Расскажите, как работает сборщик мусора.

   Это поток, работающий в фоновом режиме и непрерывно проверяющий дерево ссылок в поисках объектов, на которые больше никто не ссылается. Обнаружив такой объект, сборщик мусора освобождает занятую им память для дальнейшего использования.

5. Что такое члены класса? Назовите четыре типа членов.

   Член класса или структуры — это его часть, которая служит для хранения данных или реализует некоторую функциональность. К основным видам членов относятся поля, свойства, методы и события.

6. Что такое конструкторы и деструкторы и для чего они нужны?

   Конструктор — это метод, инициализирующий класс или структуру. При создании экземпляра класса или структуры этот метод исполняется первым. Конструктор присваивает членам класса или структуры значения по умолчанию и выполняет другие действия, необходимые для создания объекта. Деструктор — это метод, который исполняется непосредственно перед освобождением занятой объектом памяти

сборщиком мусора. В деструктор помешают код, выполняющий необходимую «очистку» перед уничтожением объекта.

7. Кратко поясните различия между уровнями доступа, которые задают для пользовательских типов и членов модификаторы *Public* (*public*), *Friend* (*internal*) и *Private (private)*.

   Экземпляры пользовательских типов, объявленных с модификатором *Public* (*public*), можно создавать в любой части приложения, тогда как экземпляры типа с модификатором *Friend* (*internal*) могут быть созданы лишь членами из сборки, в которой объявлен данный тип. Экземпляры классов с уровнем доступа *Private (private)* создают только они сами или их дочерние классы. Аналогично члены, объявленные с модификатором *Public* (*public*), могут вызывать любые клиенты приложения, члены с модификатором *Friend* (*internal*) доступны только членам из той же самой сборки, а члены с уровнем доступа *Private (private)* допускается вызывать только из типов, в которых они объявлены,

8. Нужно ли создавать экземпляр класса для обращения к общему члену? Почему?

   Нет, поскольку общие члены принадлежат типу в целом, а не отдельному экземпляру.

Глава 2. Создание пользовательского интерфейса

1. Крупный банк заказал вам приложение, которое должно быть легким в освоении, бесшовно интегрироваться с Microsoft Office XP и иметь интерфейс, дизайн которого соответствует стилю банка (для воспитания у сотрудников банка корпоративной культуры). Как достичь поставленных целей? Приведите два способа создания интерфейса.

   Попробуйте спроектировать для этого приложения интерфейс в стиле Microsoft Office XP, чтобы пользователи этого пакета комфортно чувствовали себя при работе с новым приложением. Интегрировав логотип банка и другие элементы фирменного стиля в интерфейс приложения, вы сможете выполнить поставленную заказчиком задачу.

2. Вы создаете приложение, несколько форм которого имеют идентичный набор элементов управления. Как быстрее всего создать такие формы?

   Создайте родительскую форму с общим набором элементов управления и используйте ее как основу для создания производных форм со специфической функциональностью при помощи наследования.

3. Каждый раз, когда форма получает фокус, должно отображаться приглашение на ввод значения. Опишите наилучшую стратегию для реализации подобной функциональности.

   Поместите соответствующий код в обработчик события *Activated*.

4. Как задать порядок передачи фокуса между элементами управления на форме по нажатию клавиши Tab? Приведите два способа решения этой задачи.

   Это возможно средствами Visual Studio, выбрав в меню View элемент Tab Index и щелкнув элементы управления в порядке желаемой очередности передачи фокуса. Альтернативный способ — установить свойство *TabIndex* программно или с помощью окна Properties.

5. Что такое провайдеры дополнительных свойств и для чего их применяют?

   Провайдеры дополнительных свойств — это компоненты, такие, как *ErrorProvider*, *HelpProvider* и *ToolTip*, предоставляющие размещенным на форме элементам управления дополнительные свойства. Эти компоненты применяют для отображения

пользователям дополнительных сведений об элементах управления, составляющих интерфейс приложений.

6. Для каких целей лучше использовать контекстное меню?

   Если все возможности программы доступны через главное меню, контекстное меню может стать излишним и только затруднять работу с программой. Однако контекстные меню позволяют оперативно получать доступ к возможностям, необходимым только в определенных ситуациях и только там, где это требуется,

7. Объясните, что такое «проверка на уровне формы» и «проверка на уровне поля».

   Под первым процессом понимают одновременную проверку всех данных, введенных в форму, перед ее отправкой на сервер или локальному приложению, а под вторым — поочередную проверку всех полей формы данных по мере их заполнения.

Глава 3. Типы и члены

1. Объясните, в каких случаях преобразование типов выполняется неявно, а в каких — явно. В чем опасность явного преобразования типов?

   Неявное преобразование разрешается, если оно не угрожает потерей данных, в противном случае требуется выполнить явное преобразование. Учтите, что при явном преобразовании данных возможно снижение точности значения переменной и генерация исключения.

2. Почему вместо значений лучше использовать в коде перечисления и константы?

   Применение перечислений и констант делает код более понятным и облегчает его сопровождение, позволяя заменить часто используемые значения метками, понятными человеку.

3. Кратко опишите сходства и различия между массивами и наборами.

   Массивы и наборы применяют для управления группами объектов. Чтобы получить доступ к нужному объекту, хранящемуся в массиве или наборе, достаточно указать индекс этого объекта. Для обработки членов массивов (и большинства наборов) применяют циклы *For Each...Next* (*foreach*). Размеры массивов фиксированы, их члены необходимо инициализировать перед использованием. Члены набора необходимо объявить и инициализировать за пределами набора, а потом явно добавить к набору. Классы наборов из пространства имен *System.Collections* могут динамически уменьшаться и увеличиваться; кроме того, добавлять и удалять элементы наборов разрешается во время выполнения.

4. Чем свойства отличаются от полей? Почему для представления доступа к открытым данным лучше применять свойства, а не поля?

   При обращении к свойству для чтения или установки исполняется код, проверяющий допустимость запрошенной операции — это дает разработчику больший контроль над чтением и записью свойств. В отличие от свойств, при чтении и установке полей никакой проверки не выполняется.

5. Объясните, что такое делегат, и расскажите, как он работает.

   Делегат работает как строго типизированный указатель на функцию. При помощи делегатов можно вызывать методы, на которые ссылаются делегаты, не прибегая к явному вызову этих методов.

Глава 4. Объектно-ориентированное программирование и полиморфизм

1. Дайте краткое определение концепции инкапсуляции и раскройте ее значение для ООП.

   Принцип инкапсуляции гласит, что все необходимые данные и код, реализующий функциональность объекта, должны быть заключены внутри объекта. Благодаря инкапсуляции объекты представляют собой независимые взаимозаменяемые функциональные единицы, которые можно модифицировать независимо от других блоков кода.

2. Что такое перегрузка и для чего она применяется?

   Перегрузка позволяет создать несколько методов с одинаковыми именами, но с разными сигнатурами. Перегрузку удобно применять для создания методов с идентичной или близкой функциональностью, принимающих разные наборы параметров.

3. Вам необходимо создать несколько не связанных между собой классов, предоставляющих общий набор методов. Как, используя принцип полиморфизма, сделать методы, предоставляемые этими классами, доступными другим классам?

   Выделите общий для всех классов набор методов и создайте интерфейс, предоставляющий эти методы. После этого реализуйте созданный интерфейс в каждом классе. Это позволит неявно преобразовать любой из этих классов к типу интерфейса, после чего этот класс сможет взаимодействовать с остальными классами.

4. Вам нужно создать несколько классов, поддерживающих основные возможности, но предназначенных для взаимодействия с разными наборами объектов. Какой способ выбрать, чтобы затратить на разработку таких классов минимальное время?

   Создайте класс, реализующий все функции, общие для этих классов. Затем объявите на его основе производные классы и реализуйте в них дополнительные функции, уникальные для каждого из них.

5. Что такое абстрактные классы и для чего их применяют?

   Абстрактные классы — это особая разновидность классов, не допускающих создания экземпляров, эти классы можно использовать только как базовые. У абстрактного класса имеются как реализованные, так и абстрактные методы; последние применяют в производных классах. Так, объявляя абстрактный класс, для общих методов можно предоставить полную реализацию, а для уникальных — только интерфейс, оставив их реализацию на долю производных классов.

Глава 5. Тестирование и отладка приложений

1. Опишите режим пошагового исполнения кода и доступные в нем возможности.

   Режим пошагового исполнения обеспечивает построчное исполнение кода, позволяя анализировать поведения программы во время выполнения. Предусмотрено несколько режимов пошагового исполнения, как-то: Step Into, Step Over, Step Out, Run To Cursor и Set Next Statement.

2. Для чего служит окно Watch?

   Окно Watch применяют для наблюдения за переменными приложения в режиме пошагового исполнения.

3. Представьте, что вы тестируете бета-версию большого приложения. Вам нужно регистрировать во время выполнения сведения о производительности приложения в текстовых файлах. Кратко опишите стратегию решения этой задачи.

   Разместите в коде приложения операторы *Trace,* генерирующие данные, которые вас интересуют. Затем создайте объект *TextWriterTraceListener* и добавьте его к набору *Listeners.* После этого создайте трассировочные переключатели, управляющие операторами *Trace,* и настройте объект *TextWriterTraceListener* для записи трассировочных данных в текстовый файл. Скомпилируйте приложение, установите его и включите операторы *Trace,* активировав трассировочные переключатели.

4. Нужно ли при тестировании метода проверять обработку заведомо недопустимых аргументов? Почему?

   Да, нужно. Помимо тестирования обработки допустимых аргументов следует проверить, как программа обрабатывает заведомо недопустимые значения, чтобы убедиться в ее способности устранять последствия ошибок ввода без катастрофических сбоев.

5. Кратко опишите назначение каждой части конструкции *Try...Catch...Finally* (*try...catch...finally*).

   В блок *Try (try)* заключают код приложения, который может генерировать исключение. Если при исполнении этого кода возникает исключение, оно перехватывается соответствующим блоком *Catch* (*catch*), содержащим код, способный обработать это исключение. В завершение управление передается блоку *Finally* (*finally*), который исполняется независимо от того, обработано исключение или нет.

Глава 6. Доступ к данным с помощью ADO.NET

1. Перечислите основные компоненты провайдера данных и назовите их функции.

   Провайдер данных ADO.NET — это набор компонентов, обеспечивающих доступ к данным. Любой провайдер данных состоит по крайне мере из трех объектов: *Connection* представляет реальное соединение с источником данных, *Command* — команду, исполняемую над источником, *DataReader* обеспечивает доступ для однонаправленного последовательного чтения к присоединенному источнику данных и *DataAdapter* предназначен для организации доступа к отсоединенным данным.

2. Кратко опишите особенности доступа к данным в ADO.NET с использованием постоянного подключения и отсоединенных данных.

   В ADO.NET доступ к присоединенным данным осуществляется посредством объекта *DataReader* — упрощенного объекта, обеспечивающего очень быструю и эффективную выборку данных. Однако его возможности значительно ограничены: он способен читать записи только в одном направлении, не позволяет их модифицировать и требует монопольного доступа к объекту *Connection.* В отличие от *DataReader,* объект *DataAdapter,* применяемый для доступа к отсоединенным данным, осуществляет выборку и модификацию данных при помощи соответствующих команд. Так, для выборки данных *DataAdapter* исполняет над базой данных команду SELECT: он открывает соединение, чтобы извлечь данные, загружает их в объект *DataSet,* представляющий копию данных в памяти, и сразу же закрывает соединение. Завершив обработку данных, провайдер данных обновляет содержимое источника, генерируя соответствующие команды. При этом *DataAdapter* поддерживает соединение открытым только во время исполнения команд над базой данных.

3. Какие значения принимает свойство *CommandType* объектов *SqlCommand* или *OleDbCommand* и что определяют эти значения?

   Свойство *CommandType* объекта *Command* может принимать значения *Text*, *StoredProcedure* или *TableDirect*. В первом случае строка, хранящаяся в свойстве *CommandText* объекта *Command*, интерпретируется как оператор SQL; если *CommandType* = *StoredProcedure* объект команды вызывает из базы данных хранимую процедуру и возвращает результаты, а если *CommandType* = *TableDirect*, команда возвращает все содержимое таблицы, заданной свойством *CommandText*.

4. Как в ADO.NET выполнить команду DDL, например ALTER или CREATE TABLE?

   Для исполнения команд DDL применяют объекты *Command*, При этом свойство *CommandType* объекта *Command* следует установить в *Text*, а саму команду поместить в свойство *CommandText*. После этого для исполнения команды остается вызвать метод *Command.ExecuteNonQuery*.

5. Кратко опишите преимущества и недостатки применения типизированных объектов *DataSet*.

   Типизированные объекты *DataSet* позволяют обращаться с данными, как с объектами обшей системы типов .NET. Благодаря этой особенности приложение «знает» тип данных, возвращаемых объектом *DataSet*, что исключает ошибки, возникающие из-за некорректного преобразования типов, поскольку все они обнаруживаются во время компиляции. Нетипизированные объекты *DataSet* применяют, когда структура данных не известна заранее; они также удобны тем, что их можно применять с любым источником данных.

6. Опишите механизм управления потоком данных в форме, содержащей несколько элементов управления, связанных с данными?

   С каждым источником данных, предоставляющим отображаемые на форме значения, связан объект *CurrencyManager*, отслеживающий положение записи, отображаемой в данный момент связанным элементом управления. Для удобства все объекты *CurrencyManager*, связанные с формой, доступны через ее свойство *BindingContext*. Чтобы перейти к следующей записи источника данных, достаточно изменить Свойство *Position* объекта *CurrencyManager*.

7. Расскажите, как с помощью объекта *DataView* выполнить фильтрацию и упорядочить данные.

   Чтобы задать критерий упорядочения содержимого объекта *DataView*, запишите в его свойство *Sort* имя столбца (столбцов), по которому необходимо упорядочить данные. Чтобы задать фильтрацию содержимого объекта *DataView*, запишите в его свойство *RowFilter* выражение, определяющее допустимый критерий фильтрации.

8. Дайте краткую характеристику объекту *XmlDataDocument*. Как он связан с объектом *DataSef*.

   Объект *XmlDataDocument* представляет загруженные в память данные в формате XML, организованные в иерархическую структуру. Каждый объект *XmlDataDocument* синхронизируется с объектом *DataSef*, изменения, внесенный в один объект, автоматически реплицируются на другой, поэтому при помощи *XmlDataDocument* можно манипулировать содержимым объекта *DataSet* в формате XML.

Глава 7. Создание элементов управления с помощью .NET Framework

1. Кратко опишите каждый из трех видов нестандартных элементов управления и укажите их отличия.

   Имеется три типа нестандартных элементов управления, создаваемых пользователем: производные, составные и формируемые. Производные элементы управления объявляют на основе стандартных элементов управления Windows Forms, от которых они наследуют облик и функциональность. Составные элементы управления получают путем объединения стандартных элементов управления Windows Form и добавления отсутствующих у них функций.

   Формируемые элементы управления объявляют на основе класса *Control*. Разработка этого типа элементов управления требует больше всего времени, поскольку они наследуют лишь общую функциональность, а код для прорисовки, как и все специфические возможности, должен написать разработчик.

2. Расскажите о роли объектов *Graphics*, *Brush*, *Pen* и *GraphicsPath* в визуализации графических элементов.

   Объект *Graphics* представляет поверхность для рисования и инкапсулирует методы для отображения на ней графических элементов. Объект *Brush* применяют для рисования закрашенных фигур, а *Pen* — для рисования контуров. Объект *GraphicsPath* представляет сложные фигуры, визуализируемые при помощи объекта *Graphics*.

3. Как нарисовать текст в клиентской области формы или другого объекта?

   Прежде всего необходимо получить ссылку на объект *Graphics*, затем создать экземпляр объекта *GraphicsPath*. Далее при помоши метода *GraphicsPath.AddString* следует добавить к *GraphicsPath* нужный текст — осталось вызвать *Graphics.DrawPath* или *Graphics.FillPath* для визуализации текста.

4. Какова роль объекта *License Provider* в лицензировании элементов управления?

   Этот объект управляет проверкой лицензий во время выполнения и предоставляет доступ к лицензируемым компонентам, для которых имеется действительная лицензия. Метод *License Manager. Validate* проверяет наличие файла лицензии; обнаруженная лицензия проверяется по алгоритму, заданному объектом *License Provider*, Используемый объект *LicenseProvider* задают при помощи атрибута *LicenseProviderAttribute*.

5. Как создать форму или элемент управления, форма которого будет отличаться от прямоугольной?

   Занесите в свойство *Region* формы одноименный объект, представляющий фигуру желаемой формы. Такой объект *Region* можно создать на основе объекта *GraphicsPath*.

Глава 8. Сложные случаи программирования с применением .NET Framework

1. Расскажите, как вывести на печать документ при помощи компонента *PrintDocument*, выдержав межстрочные интервалы и, если это многостраничный документ, напечатать все его страницы?

   У класса *PrintDocument* есть метод *Print*, генерирующий событие *PrintPage*. Код для прорисовки печатаемого содержимого на принтере следует помещать в обработчик этого события. Последний передает подлежащие печати объекты через эк-

земпляр класса *PagePrintEventArgs*.Прорисовка на принтере выполняется посредством объекта *Graphics,* который также перелается через объект *PagePrintEventArgs,* Чтобы рассчитать верный межстрочный интервал, разделите высоту прямоугольника, заданного свойством *MarginBounds*, на кегль шрифта, которым набран текст. Для печати многостраничного документа установите свойство *PagePrintEventArgs.HasMorePages* в *true* — это приведет к повторной генерации события *PrintPage.* У обработчика события *PrintPage* нет параметров для хранения числа напечатанных страниц, поэтому код, контролирующий вывод на печать многостраничных документов, необходимо добавить к обработчику самостоятельно.

2. Как с помощью методов *Begin* и *End* асинхронно вызвать метод Web-сервиса?

   Любой открытый метод Web-сервиса можно вызвать как синхронно, так и асинхронно. Для асинхронного вызова следует вызвать метод *Begin<webmethod>*, где *<webmethod>* — имя метода. Этому методу необходимо передать делегат, ссылающийся на соответствующий метод обратного вызова и возвращающий объект типа *IAsyncResult*, который передается как параметр методу обратного вызова. Чтобы получить реальные результаты Web-метода, следует вызвать метод *End<webmethod>*, передав ему ссылку на объект, возвращенный методом *Begin<webmethod>*.

3. Перечислите и кратко поясните требования, необходимые для реализации специальных возможностей доступа, которые предъявляются к программам—кандидатам на получение логотипа «Certified for Windows».

   Этих требований пять:

   - поддержка стандартных параметров пользовательского интерфейса. Так, цвет, шрифт и другие параметры оформления пользовательского интерфейса приложения должны соответствовать общесистемным параметрам;
   - поддержка высококонтрастных цветовых схем (режим High Contrast). Чтобы приложение соответствовало этому требованию, для оформления интерфейса следует применять только цвета системной палитры;
   - наличие документированных способов доступа с клавиатуры ко всем возможностям интерфейса. Основные средства реализации этого требования — применение быстрых клавиш и создание хорошей документации на приложение;
   - указание текущего положения фокуса ввода — реализуется в основном средствами .NET Framework;
   - поддержка различных способов отображения информации, помимо звуковых сигналов. Соответствие этому требованию достигается организацией поддержки избыточных средств отображения информации.

4. Расскажите, как при помощи компонента *HelpProvider* снабдить элементы пользовательского интерфейса справочными сообщениями.

   Эта задача решается с помощью компонента *HelpProvider*, способного отображать всплывающие подсказки (для этого используется значение свойства *HelpString* элементов управления) и разделы справочной системы. *HelpProvider* предоставляет всем размещенным на форме элементам управления дополнительные свойства *HelpString, HelpKeyWord* и *HelpNavigator.* Если свойство *HelpProvider.HelpNameSpace* не установлено, справочная информация извлекается из свойства *HelpString* и отображается как всплывающая подсказка, в противном случае по нажатию Г1 *HelpProvider* отображает для данного элемента управления раздел справочной системы, заданный свойствами *HelpKeyWord* и *HelpNavigator.*

5. Как создать локализованную версию формы?

   Для этого необходимо установить свойство *Localizable* формы в *true*. Далее в свойство *Language* следует записать код языка и/или региона, для которого предназначена локализованная форма. Затем следует внести в интерфейс необходимые изменения, связанные с локализацией. Значения всех свойств, измененных в ходе локализации, автоматически записываются в ресурсные файлы и загружаются, когда свойству *CurrentUICulture* присваивается соответствующее значение *CultureInfo*.

6. Как преобразовать данные из унаследованной кодировки в формат Unicode?

   Это можно сделать при помощи метода *Encoding.Convert*. Он требует объекты, представляющие исходную и целевую кодировки, а также массив байт с данными, подлежащими преобразованию. *Encoding.Convert* возвращает массив байт, преобразованный в целевую кодировку. Преобразовать строку или символьный массив в массив байтов можно методом *Encoding.GetBytes*, выполнить обратное преобразование позволяет метод *Encoding.GetChars*.

7. Чем отличаются глобализация и локализация?

   Глобализацией называют форматирование имеющихся данных согласно правилам, принятым в той или иной культуре, а локализацией — создание и загрузку различных версий ресурсов для каждой из поддерживаемых культур.

Глава 9. Конфигурирование и защита сборок

1. Как подписать сборку с помощью строгого имени и для чего это нужно?

   Чтобы подписать сборку с помощью строгого имени, необходим файл с ключом (если нет готового, его следует создать с помощью утилиты sn.exe). Файл с ключом нужно указать в файле AssemblyInfo, а также проверить, правильно ли задан номер версии. Во время компиляции к сборке будет добавлена цифровая подпись. Строгое имя нужно не только для идентификации сборки и управления ее версиями, но и для установки ее в GAC.

2. Расскажите, как извлечь ресурсы во время выполнения.

   Начать следует с создания экземпляра класса *ResourceManager*, связанного со сборкой, содержащей нужные ресурсы. После этого можно извлекать ресурсы в виде строк (при помощи метода *GetString*) и объектов (посредством метода *GetObject*).

3. Как во время выполнения извлечь информацию из конфигурационного файла? А как записать в него нужные данные во время разработки?

   Сначала необходимо создать объект *AppSettingsReader* и, вызвав его метод *GetValue*, прочитать содержимое конфигурационного файла. Чтобы добавить к конфигурационному файлу новые элементы, следует добавить к узлу <appSettings> элементы <add> и указать в них значение параметра с ключом, необходимым для извлечения параметра. Значения параметров можно изменять, редактируя конфигурационный файл. Чтобы коррективы вступили в силу, приложение нужно перезапустить.

4. Вы создали решение, которое должно быть доступным только членам групп Developers и Administrators локального компьютера. Как реализовать такую схему безопасности?

   Следует создать пару объектов *PrincipalPermission*, представляющих группы Developers и BUILTIN\Administrators. Третий объект *PrincipalPermission* получается при объединении первых двух вызовом метода *Union*. Для решения задачи остается потребовать полученное разрешение посредством метода *Demand*.

5. Кратко перечислите различия между декларативной и императивной моделями зашиты при использовании защиты по правам доступа к коду.

   Императивная защита реализуется посредством вызова методов объектов *Permission* во время выполнения. Декларативная защита создается путем назначения классам и методам атрибутов, представляющих разрешения. Императивная защита обеспечивает более тонкий контроль над запросами разрешений во время выполнения, зато атрибуты декларативной защиты добавляются в метаданные сборок, что позволяет анализировать их при помощи соответствующих классов из пространства имен *System.Reflection*. Кроме того, при использовании декларативной защиты с помощью директивы *Assembly (assembly)* удается запрашивать разрешения на уровне сборки.

Глава 10. Развертывание

1. Как выполнить развертывание приложения с помощью XCOPY? В каких случаях его можно применять, а в каких — нет?

   Применение DOS-команды XCOPY — простейший способ развертывания приложений путем копирования каталога приложения со всем его содержимым на целевой компьютер, однако годится он только для развертывания приложений, не использующих разделяемых файлов и не требующих выполнения специальных действий во время установки.

2. Согласно желанию заказчика приложение должно устанавливаться на компьютеры персонала через сеть, при этом для всех сотрудников его следует установить в один и тот же каталог. Как решить эту задачу?

   Прежде **всего** создайте **проект установочной программы для приложения. Далее с помощью редактора пользовательского интерфейса создайте диалоговое окно, позволяющее выбирать каталог для установки приложения только администратору и не позволяющее сделать это рядовым пользователям. После этого установите приложение на сетевой диск с использованием административной установки.**

3. Документация на ваше приложение создана в виде набора HTML-файлов. Как включить ее в состав дистрибутива приложения? Назовите минимум два способа.

   Включить несжатые HTML-файлы в дистрибутив приложения можно во время создания проекта установочной программы посредством мастера Setup либо позже, с помощью редактора файловой системы.

4. Что такое прекомпилированный образ приложения? Как его создать?

   Это образ .NET-сборки, скомпилированной в машинные коды. Создать его позволяет утилита Ngen.exe.

5. Что делает программа, подготавливающая целевой компьютер к развертыванию приложения (bootstrapper)? В каких случаях она необходима?

   Перед установкой приложения эта программа автоматически определяет наличие Windows Installer на целевой машине и, если таковой отсутствует, устанавливает его. Программа для первоначальной подготовки не требуется, если предполагается развертывать приложение на компьютерах под управлением Windows XP (в этой ОС Windows Installer 1.5 уже есть) либо на компьютерах с предустановленным Windows Installer.

Предметный указатель

.NET Framework 1. 2
— PrintDocument 344
— библиотека
— — базовых классов 2. 6, Я, 14
— — типов 357
— культура 378
— -код 373
— межъязыковая совместимость 2
— поддержка печати 343
— провайдер данных 7
— пространство имен 6
— тип данных 100 *см. также* тип
— элемент управления 320

A

ActiveX 358
ADO.NET 236-239. 258, 279. 291
API 363

C

CLR (Common Language Runtime) 1, 2, 3, 4, 394, 425
CLS (Common Language Specification) 3
collection *см.* набор
COM 357
CTS (common type system) 2, 3
Custom Actions Editor *см.* редактор, нестандартных действий

D

Data Adapter Configuration Wizard 255
DDL (Database Definition Language) 243
DLL 3, 4, 363
docking *см.* элемент управления, стыковка
DOM (Document Object Model) 294

F

Fife System Editor *см.* редактор, файловой системы
File Types Editor *см.* редактор, типов файлов

G

GAC (Global Assembly Cache) 334, 402, 403
garbage collection *см.* сбор мусора
GDI (Graphic Device Interface) 308
GDI+ 7, 308
getter *см.* метод, получатель

H

heap *см.* куча
HelpProvider 371
hide *см.* член, сокрытие

I

IL (Intermediate Language) 3, 4
indexer *см.* индексатор
inheritance *см.* форма, наследование
Inheritance Picker 46

L

Launch Conditions Editor *см.* редактор, условий

M

manifest *см.* сборка, декларация
member *см.* член
module *см.* сборка, модуль

N

namespace *см.* пространство имен
nested type *см.* тип, пложённый

O

object model *см.* обьектная модель
OLE 7
override *см.* член, переопределение

P

permission *см.* разрешение

R

reference type *см.* тип, ссылочный
Registry Editor *см.* редактор, реестра
run-timeerror *см.* ошибка, периода выполнения

S

satellite assembly *см.* сборка, сопутствующая
scope *см.* область видимости
sealed *см.* класс, запечатанный
security principal *см.* участник безопасности
setter *см.* метод, установщик
Setup Project 435
shadow *см.* член, замещение
Solution Explorer 359
SQL 253
SQLServer 7, 291
stack *см.* стек
strong name *см.* сборка, строгое имя
syntax error *см.* ошибка, синтаксическая

T

test cases *см.* тестирование, контрольный пример
Trace Listener *см.* трассировка, слушатель
transaction *см.* транзакция

U

UDDI (Universal Description Discovery Integration) 359
User Interface Editor *см.* редактор, пользовательского интерфейса

W

Web-сервис 359
Windows Forms 47, 58, 365, 367

X

XCOPY 434
XML 291, 292
XSL 296, 297

А

абстрагирование 152

В

внешний ключ 269

Д

данные
— выборка 256
— доступ 236, 240, 248
— извлечение 256, 257, 291
— источник 240, 254
— манипулирование 243
— модификация 270, 289
— набор 237
— обмен 254
- обновление 262, 270, 272, 274
— поток 282
— потребитель 279
— предварительный просмотр 258
— преобразование 296
- провайдер 237, 278, 291 .
— просмотр 278
— связывание 278, 279
— — в период выполнения 281
— — простое 279
— — сложное 279, 285
— сортировка 285, 286
— типизированные 249
— фильтрация 285, 286
— формат 264
декларативная безопасность 418
делегат 132
деструктор 24
дизайнер 45, 59, 325

Ж

журнал событий 209
«жучок» 194

И

индексатор 65, 118, 128
инкапсуляция 154
интерфейс 40, 154, 161, 163
— графических устройств *см.* GDI
— дизайн 41
— культура 376
— определение 161
— реализация 164
— член 165
исключение 195, 220, 275
— генерация 225
— — повторная 225
— класс 228
— нестандартное 227, 228
— обработка 220, 426
— обработчик 221

К

класс 8, 14, 18, 28, 152
— абстрактный 176, 178
— базовый 168, 170, 172
— запечатанный 169, 313
— производный 168, 169
— создание 169, 177
ключевое слово
— *abstract* 177, 179
— *AddHandler* 135, 137
— *base* 174
- *ByRef* 22
- *ByVal* 22
- *Class* 14, 15
— *Declare* 363
— *DESC* 287
- *End Class* 15
— *extern* 363
— *Friend* 26
— *Implements* 164, 166, 169
— *Inherits* 169
— *interface* 161, 163
— *Interface* 161, 163
— *internal* 26
— *Lib* 363
— *Me* 51
- *MustInherit* 177, 179
— *MustOverride* 177, 179
- *MyBase* 174
— *new* 172
— *New* 9, 16, 152
- *NotInheritable* 170, 321
— *Optional* 24
-*out* 23
- *Overridable* 171, 177
— *Overrides* 25
— *private* 175
— *Private* 26, 175
— *protected* 175
— *Protected* 175
- *Public* 15, 26
- *Readonly* 162, 178
— *ref* 22
— *RemoveHandler* 138
— *sealed* 170, 321
— *Shadows* 172
— *Shared* 30
— *static* 363
— *Static* 21
— *struct* 15
— *Structure* 15
— *this* 51
- *Value* 124
-*virtual* 171, 177
- *WithEvents* 135
- *WriteOnly* 162, 178
компилированный образ 458
компиляция 4
компонент 58
константа ПО
конструктор 9, 24, 226
контейнер 61
криптографический ключ 402
куча 7, 8
кэш глобальных сборок *см.* GAC

М

манипулирование строками 107
массив ПО, 113. 121
— многомерный 116
— — невыровненный 116
— — прямоугольный 116
меню 41, 71
— активация команд 75
— деактивация команд 75
— клавиши
— — быстрые 73
— — доступа 73
— клонирование 76
— контекстное 74
— переключатель 75
— разделитель 72
— слияние 77
— событие 74
— создание 71
— флажок 75
метаданные типов 394
метод 19, 30. 153
— вызов 20
— объявление 19
— параметр 21, 23
— перегруженный 157
— переменная 20
— получатель 162, 178
— статический 205
— установщик 162, 178
— формы 51
модификатор доступа 26, 29, 363

Н

набор 62, 117, 121
наследование 168, 178

О

область видимости 8, 20, 26
общая система типов *см.* CTS
общеязыковая исполняющая среда *см.* CLR
общеязыковая спецификация *см.* CLS
объект 152, 153, 154
— ограничения 270
— типизированный 258
объектная модель 153
объектно-ориентированное программирование *см.* ООП
ограничение 268
ООП (объектно-ориентированное программирование) 2, 151
отладка 194, 215
— инструмент 200
ошибка 194
— логическая 195
— обработка 274
— периода выполнения 194. 195, 220
— поиск 195
— синтаксическая 194, 230

П

панель инструментов 41
перечисление ПО, 111
поле 153
полиморфизм 154, 155, 161, 163, 168
приложение 3
— глобализация 373. 375
— динамические свойства 407
— конфигурационный файл 405. 406, 428
— локализация 375
— оптимизация 410
— — на этапе разработки 410
— — средствами компилятора 411
— — стратегия 410
— отладка 230
— развертывание 434
пространство имен 6
— *System* 6
— *System* 6
— *System. Collections* 6, 117, 120
— *System.ComponentModel* *1*
— *System. Data* *6, 1*
— *System.Data.Common* *7*
— *System.Data.OleDb* *7*
— *System.Data.SqlClient* *1*
— *System.Data.SQLClient* 6
- *System.Drawing* *1,* 308, 309, 312, 313
— *System.Drawing.2D* 308
— *System. Drawing. Design* 308
— *System.Drawing.Drawing2D* 313
— *System. Drawing.Imaging* 308
— *System. Drawing. Printing* 309
— *System. Drawing. Text* 309
— *System. IO* 7, 264
— *System.Math* *7*
— *System.Reflection* *1*
— *System.Security* *7*
— *System. Threading* *7*
— *System. Windows.Forms* 6, 7
процедура 25

Р

разделитель 264
разрешение 414
редактор
— нестандартных действий 448, 455
— пользовательского интерфейса 448, 453
— реестра 448, 450
— типов файлов 448, 451
— условий 448
— условий установки 457
— файловой системы 448
роль 414, 416, 418

С

сбор мусора 8, 32
сборка 3, 357, 394
— декларация 3, 4, 394
- закрытая 401, 402
— идентификационные данные 394
— модуль 3
— разделяемая 401, 402
— разрешение 395
— ресурсная 396, 398
— создание 395
— сопутствующая 399, 401
— строгое имя 402
— установка 458

— файл 394
— — ресурсов 398
— централизованное хранение 402
свойство 123, 153
— набор 129
— параметризованное 127
— по умолчанию 128
— создание 125, 126, 127, 129
— только для записи 126
— только для чтения 126
событие 53, 132, 153
— генерация 133
— группа 136
— меню 74
- обработчик 53, 67, 132, 135, 136
— объявление 133
справочная система 370
стек 7
структура 8, 15. 16, 18. 28, 310

Т

тестирование 215
— блочное 215
— — план 215
— данные 217
— контрольный пример 216
тип 4, 100
— булев 8
— вложенный 16, 29
— встроенная функциональность 106
— значений 7, 8, 16
— — с плавающей точкой 101
— метод 106
— модификатор доступа 28
— нечисловой 102
— объектно-ориентированный 2
— перечислимый 8
— пользовательский 16
— преобразование 103
— — неявное 103
— — явное 104
— приведение 104
— распаковка 106
— символьный 8
— ссылочный 7, 8, 9, 11, 16, 152
— упаковка 106
— целочисленный 8, 100
точка прерывания 199
— отключение 200
— удаление 200
— управление 200
транзакция 272
трассировка 205
— запись данных 208, 209
— переключатель 210, 212
— слушатель 205

У

участник безопасности 415

Ф

фокус ввода 83
форма 40, 45, 85, 86
— видимая 52, 56
— внешний вид 49
— добавление 45
— жизненный цикл 55
— закрытие 56
— композиция 42
— локализованная 376
— метод 51
— наследование 46
— невидимая 53, 56
— освобождение ресурсов 53
— прозрачность 50
— размещение 48
— свойство 48, 49
— событие 53, 55
— создание 383
— стартовая 47, 48
— текст 50
— фокус 52, 56
— цвет 50
— экземпляр 55
функция 25

Ч

член 14, 153, 165
— абстрактный 177
— добавление 15
— доступ 174
— замещение 170, 172
— защищенный 175
— наследуемый 170
— переопределение 170, 171
— сокрытие 170, 172

Э

экземпляр 9, 16
элемент управления 40, 41, 42, 58, 320
-ActiveX 320, 358
— Windows Forms 367
— вложенный 61
— граница 310
— добавление 59
— — на панель 330
— — членов 322
— значок 330
— лицензирование 332
— модификация внешнего вида 324
— набор 65
— наследование 321, 322
— нестандартный 66
— обработчик события 67
— объединение 61
— объявление 321
— отладка 331, 332
— пользовательский 321
— провайдер дополнительных свойств 68
— производный 323, 324
— редактирование свойств 59
— создание 320
— составной 321, 325
— стыковка 62, 64
— тестирование 340
— фиксация 62
— фокус 60
— формируемый 322, 327

ЛИЦЕНЗИОННОЕ СОГЛАШЕНИЕ MICROSOFT

прилагаемый к книге компакт-диск

ЭТО ВАЖНО — ПРОЧИТАЙТЕ ВНИМАТЕЛЬНО. Настоящее лицензионное соглашение (далее «Соглашение») является юридическим документом, оно заключается между Вами (физическим *или* юридическим лицом) и Microsoft Corporation (далее «корпорация Microsoft») на указанный выше продукт Microsoft, который включает программное обеспечение и может включать сопутствующие мультимедийные и печатные материалы, а также электронную документацию (далее «Программный Продукт»). Любой компонент, входящий в Программный Продукт, который сопровождается отдельным Соглашением, подпадает под действие именно того Соглашения, а не условий, изложенных ниже. Установка, копирование или иное использование данного Программного Продукта означает принятие Вами данного Соглашения. Если Вы не принимаете его условия, то не имеете права устанавливать, копировать или как-то иначе использовать этот Программный Продукт.

ЛИЦЕНЗИЯ НА ПРОГРАММНЫЙ ПРОДУКТ

Программный Продукт защищен законами Соединенных Штатов по авторскому праву и международными договорами по авторскому праву, а также другими законами и договорами по правам на интеллектуальную собственность.

1. **ОБЪЕМ ЛИЦЕНЗИИ.** Настоящее Соглашение лает Вам право:
   a) **Программный** продукт. Вы можете установить и использовать одну копию Программною Продукта на одном компьютере. Основной пользователь компьютера, на котором установлен данный Программный Продукт, может сделать только для себя вторую копию и использовать ее на портативном компью гере.
   b) **Хранение или** использование в сети. Вы можете также скопировать или установить экземпляр Программного Продукта на ус гройстве хранения, например на сетевом сервере, исключительно для установки ил и запуска данного Программного Продукта на других компьютерах в своей внутренней сети, но тогда Вы должны приобрести лицензии на каждый такой компьютер. Лицензию на данный Программный продукт нельзя использовать совместно или одновременно на других компьютерах.
   c) License Pah. Если Вы купили эту лицензию в составе Microsoft License Pak, можете сделать ряд дополнительных копий программного обеспечения, входящего в данный Программный Продукт, и использовать каждую копию так, как было описано выше. Кроме того. Вы получаете право сделать соответствующее число вторичных копий для портативного компьютера в целях, также оговоренных выше.
   o) **Примеры кода.** Это относится исключительно к отдельным частям Программного Продукта, заявленным как примеры кода (далее «Примеры»), если таковые входят в состав Программного Продукта.
      i) Использование и **модификация.** Microsoft дает Вам право использовать и модифицировать исходный код Примеров при условии соблюдения пункта (d)(iii) ниже. Вы не имеете права распространять в виде исходного кода ни Примеры, ни их модифицированную версию.
      ii) Распространяемые файлы. При соблюдении пункта (d)(iii) Microsoftдает Вам право на свободное от отчислений копирование и распространение в виде объектного кода Примеров или их модифицированной версии, кроме тех частей (или их модифицированных версий), которые оговорены в файле Readme, относящемся к данному Программному Продукту, как не подлежащие распространению.
      iii) Требования к **распространению** файлов. Вы можете распространять файлы, разрешенные к распространению, при условии, что: а) распространяете их в виде объектного кода только в сочетании со своим приложением и как его часть; б) не используете название, эмблему или товарные знаки Microsoftдля продвижения своего приложения; в) включаете имеющуюся в Программном Продукте ссылку на авторские права в состав этикетки и заставки своего приложения; г) согласны освободить от ответственности и взять на себя защиту корпорации Microsoft от любых претензий или преследований по закону, включая судебные издержки, если таковые возникнут в результате использования или распространения Вашего приложения; и д) не допускаете дальнейшего распространения конечным пользователем своего приложения. По поводу отчислений и других условий лицензии применительно к иным видам использования или распространения распространяемых файлов обращайтесь в Microsoft.

2. **ПРОЧИЕ ПРАВА И ОГРАНИЧЕНИЯ**
   - **Ограничения** на **реконструкцию, декомпиляцию и дизассемблирование.** Вы не имеете права реконструировать, декомпилировать или дизассемблировать данный Программный Продукт, кроме того случая, когда такая деятельность (только в той мере, которая необходима) явно разрешается соответствующим законом, несмотря на это ограничение.

- Разделение компонентов. Данный Программный Продукт лицензируется как единый продукт. Ею компоненты нельзя отделять друг от друга для использования более чем на одном компьютере,
- Аренда. Данный Программный Продукт нельзя сдавать в прокат, передавать во временное пользование или уступать для использования в иных целях.
- Услуги по технической поддержке. Microsoft может (но не обязана) предоставить Вам услуги по технической поддержке данного Программного Продукта (далее «Услуги»). Предоставление Услуг регулируется соответствующими правилами и программами Microsoft, описанными в руководстве пользователя, электронной документации и/или других материалах, публикуемых Microsoft. Любой дополнительный программный коп. предоставленный в рамках Услуг, следует считать частью данного Программного Продукта и подпадающим под действие настоящего Соглашения. Что касается технической информации, предоставляемой Вами корпорации Microsoft при использовании ее Услуг, то Microsoft может задействовать эту информацию в деловых целях, в том числе для технической поддержки продукта и разработки. Используя такую техническую информацию, Microsoft не будет ссылаться на Вас.
- Передача прав на программное обеспечение. Вы можете безвозвратно уступить все права, регулируемые настоящим Соглашением, при условии, что не оставите себе никаких копий, передадите все составные части данного Программного Продукта (включая компоненты, мультимедийные и печатные материалы, любые обновления, Соглашение и сертификат подлинности, если таковой имеется) и принимающая сторона согласится с условиями настоящего Соглашения.
- **Прекращение действия** Соглашения. Без ущерба для любых других прав Microsoft может прекратить действие настоящего Соглашения, если Вы нарушите его условия. В этом случае Вы должны будете уничтожить все копии данного Программною Продукта вместе со всеми его компонентами.

3. АВТОРСКОЕ ПРАВО. Все авторские права и право собственности на Программный Продукт (в том числе любые изображения, фотографии, анимации, видео, аудио, музыку, текст, примеры кода, распространяемые файлы и апплеты, включенные в состав Программного Продукта) и любые его копии принадлежат корпорации Microsoft или ее поставщикам. Программный Продукт охраняется законодательством об авторских правах и положениями международных договоров. Таким образом, Вы должны обращаться с данным Программным Продуктом, как с любым другим материалом, охраняемым авторскими правами, с тем исключением, что Вы можете установить Программный Продукт на один компьютер при условии, что храните оригинал исключительно как резервную или архивную копию. Копирование печатных материалов, поставляемых вместе с Программным Продуктом, запрещается.

ОГРАНИЧЕНИЕ ГАРАНТИИ

ДАННЫЙ ПРОГРАММНЫЙ ПРОДУКТ (ВКЛЮЧАЯ ИНСТРУКЦИИ ПО ЕГО ИСПОЛЬЗОВАНИЮ) ПРЕДОСТАВЛЯЕТСЯ БЕЗ КАКОЙ-ЛИБО ГАРАНТИИ. КОРПОРАЦИЯ MICROSOFT СНИМАЕТ С СЕБЯ ЛЮБУЮ ВОЗМОЖНУЮ ОТВЕТСТВЕННОСТЬ, В ТОМ ЧИСЛЕ ОТВЕТСТВЕННОСТЬ ЗА КОММЕРЧЕСКУЮ ЦЕННОСТЬ ИЛИ СООТВЕТСТВИЕ ОПРЕДЕЛЕННЫМ ЦЕЛЯМ. ВЕСЬ РИСК ПО ИСПОЛЬЗОВАНИЮ ИЛИ РАБОТЕ С ПРОГРАММНЫМ ПРОДУКТОМ ЛОЖИТСЯ НА ВАС.

НИ ПРИ КАКИХ ОБСТОЯТЕЛЬСТВАХ КОРПОРАЦИЯ MICROSOFT, ЕЕ РАЗРАБОТЧИКИ, А ТАКЖЕ ВСЕ, ЗАНЯТЫЕ В СОЗДАНИИ, ПРОИЗВОДСТВЕ И РАСПРОСТРАНЕНИИ ДАННОГО ПРОГРАММНОГО ПРОДУКТА, НЕ НЕСУТ ОТВЕТСТВЕННОСТИ ЗА КАКОЙ-ЛИБО УЩЕРБ (ВКЛЮЧАЯ ВСЕ. БЕЗ ИСКЛЮЧЕНИЯ, СЛУЧАИ УПУЩЕННОЙ ВЫГОДЫ, НАРУШЕНИЯ ХОЗЯЙСТВЕННОЙ ДЕЯТЕЛЬНОСТИ, ПОТЕРИ ИНФОРМАЦИИ ИЛИ ДРУГИХ УБЫТКОВ) ВСЛЕДСТВИЕ ИСПОЛЬЗОВАНИЯ ИЛИ НЕВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ДАННОГО ПРОГРАММНОГО ПРОДУКТА ИЛИ ДОКУМЕНТАЦИИ, ДАЖЕ ЕСЛИ КОРПОРАЦИЯ MICROSOFT БЫЛА ИЗВЕЩЕНА О ВОЗМОЖНОСТИ ТАКИХ ПОТЕРЬ, ТАК КАК В НЕКОТОРЫХ СТРАНАХ НЕ РАЗРЕШЕНО ИСКЛЮЧЕНИЕ ИЛИ ОГРАНИЧЕНИЕ ОТВЕТСТВЕННОСТИ ЗА НЕПРЕДНАМЕРЕННЫЙ УЩЕРБ, УКАЗАННОЕ ОГРАНИЧЕНИЕ МОЖЕТ ВАС НЕ КОСНУТЬСЯ.

РАЗНОЕ

Настоящее Соглашение регулируется законодательством штата Вашингтон (США), кроме случаев (и лишь в той мере, насколько это необходимо) исключительной юрисдикции того государства, на территории которого используется Программный Продукт.

Если у Вас возникли какие-либо вопросы, касающиеся настоящего Соглашения, или если Вы желаете связаться с Microsoft по любой другой причине, пожалуйста, обращайтесь в местное представительство Microsoft или пишите по адресу: Microsoft Sales Information Center, One Microsoft Way. Redmond, WA 98052-6399.

Microsoft Corporation

Разработка Windows-приложений при помощи Microsoft Visual Basic .NET и Microsoft Visual C# .NET

Учебный курс MCAD/MCSD

Перевод с английского под общей редакцией **А. Е. Соловченко**

Редактор **Ю. П. Леонова**

Компьютерный дизайн и подготовка иллюстраций **В, Б. Хильченко**

Технический редактор **О. В. Дергачева**

Дизайнер обложки **Е. В. Козлова**

Оригинал-макет выполнен с использованием
издательской системы Adobe PageMaker 6.0

Para(-)Type

TypeMarket*FontLibrary*
легальный пользователь

Главный редактор **А. И. Козлов**

Подготовлено к печати издательством «Русская Редакция»
121087. Москва, ул. Заречная, д, 9
тел.: (095) 142-0571, тел./факс: (095) 145-4519
e-mail: info@rusedit.ru, http:// www.rusedit.ru

РУССКАЯ РЕДАКЦИЯ

Подписано в печать 20.06.2003 г. Тираж 2500 экз.
Формат 60x90 1/16. Физ п *я.* 32
Отпечатано в ОАО «Типография "Новости"»
105005, Москва, ул. Фр. Энгельса, 46

Книги Microsoft Press на русском языке по программам сертификации Microsoft

Сертификационный экзамен	Издания, необходимые для подготовки к экзамену	Программа сертификации
№70-100 Analyzing Requirements and Defining Solution Architectures	Принципы проектирования и разработки программного обеспечения. Учебный курс MCSD 2-е изд. ISBN 5-7502-0213-5, 736 стр., +CD. 2002 г.	MCSD
№ 70-210 Installing, Configuring and Administering Microsoft Windows 2000 Professional	MicrosoftWindows 2000 Professional. Учебный курс MCSA/MCSE. 3-е изд. ISBN 5-7502-0246 1. 672 стр., 2003 г.	MCSA/MCSE
№70-215 Installing, Configuring and Administering Microsoft Windows 2000 Server	Microsoft Windows 2000 Server. Учебный курс MCSA/MCSE 3-е изд. ISBN 5-7502 0219-4, 912 стр., +CD, 2002 г.	MCSA/MCSE
№ 70-216 Implementing and Administering a Microsoft Windows 2000 Network Infrastructure	Администрирование сети на основе MicrosoftWindows 2000. Учебный курс MCSE. 2-е изд. ISBN 5-7502-0233-X, 512 стр., +CD. 2003 г.	MCSA/MCSE
№70-217 Implementing and Administering a Microsoft Windows 2000 Directory Services Infrastructure	Microsoft Window» 2000 Active Directory Services. Учебный курс MCSE. 2-е изд. ISBN 5-7502-0247-X. 800 стр., 2003 г.	MCSE
№70-218 Managing a Microsoft Windows 2000 Network Environment	Управление сетевой средой Microsoft Windows 2000. Учебный курс MCSA/MCSE ISBN 5 7502-0212 7, 896 стр., 2003 г.	MCSA/MCSE
№ 70-220 Designing a MicrosoftWindows 2000 Network Security	Безопасность сети на основе Microsoft Windows 2000. Учебный курс MCSE. ISBN 5-7502-0176-7, 912 стр.. +CD, 2001 г.	MCSE
№ 70-221 Designing a Microsoft Windows 2000 Network Infrastructure	Разработка инфраструктуры сетевых служб MicrosoftWindows 2000. Учебный курс MCSE. ISBN 5 7502-0192-9, 992 стр., +CD, 2001 г.	MCSE
№70-226 Designing Highly Available Web Solutions with Microsoft Windows 2000 Server Technologies	Создание Web-решений высокой доступности на основе MicrosoftWindows 2000 Server. Учебный курс MCSE. ISBN 5-7502-0196-1, 576 стр., +CD. 2002 г.	MCSE
№ 70-227 Installing, Configuring and Administering Microsoft Internet Security and Acceleration (ISA) Server 2000	Microsoft Internet Security and Acceleration Server 2000. Учебный курс MCSE. ISBN 5-7502-0191-0, 544 стр., +CD. 2002 г.	MCSA/MCSE
№ 70-228 Installing. Configuring and Administering Microsoft SQL Server 2000 Enterprise Edition	Администрирование Microsoft SQL Server 2000. Учебный курс MCSA/MCSE/MCDBA. ISBN 5-7502-0202-X, 816стр., +CO, 2002 г.	MCSA/MCSG/MCDBA

Книги Microsoft Press на русском языке по программам сертификации Microsoft

Сертификационный экзамен	Издания, необходимые для подготовки к экзамену	Программа сертификации
№ 70-270 Installing, Configuring and Administering Microsoft Windows XP Professional	**Microsoft Windows XP Professional Учебный курс MCSA/MCSE.** 2-е изд. ISBN 5-7502-0250-X, 1008 стр., +CD, 2003 г.	**MCSA/MCSE**
№ 70-305 Developing and Implementing Web Applications with Microsoft Visual Basic .NET and Microsoft Visual Studio .NET **№ 70-315** Developing and Implementing Web Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET	**Разработка Web-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET Учебный курс MCAD/MCSD.** ISBN 5-7502-0221-6, 704 стр., +CD, 2003 г.	**MCAD/MCSD**
№ 70-306 Developing and Implementing Windows-based Applications with Microsoft Visual Basic .NET and Microsoft Visual Studio .NET **№ 70-316** Developing and Implementing Windows-based Applications with Microsoft Visual C# .NET and Microsoft Visual Studio .NET	**Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET Учебный курс MCAD/MCSD.** ISBN 5-7502-0222-4. 512 стр., +CD, 2003 г.	**MCAD/MCSD**
№ 70-300 Analyzing Requirements and Defining Microsoft .NET Solution Architectures	**Анализ требований и определение архитектуры решений Microsoft .NET. Учебный курс MCSD.** ISBN 5-7502-0248-8, готовится к изданию	**MCAD/MCSD**
№ 70-310 Developing XML Web Services and Server Components with Microsoft Visual Basic .NET and the Microsoft .NET Framework **№ 70-320** Developing XML Web Services and Server Components with Microsoft Visual C# and the Microsoft .NET Framework	**Разработка XML Web-сервисов и серверных компонентов на** Microsoft **Visual Basic .NET и Microsoft C# .NET. Учебный курс MCAD/MCSD.** ISBN 5-7502-0249-6, готовится к изданию	**MCAD/MCSD**
Network+ Certification	**Компьютерные сети. Сертификация Network+. Учебный курс** ISBN 5-7502-0190-2, 704 стр., + CD, 2002 г.	CompTIA

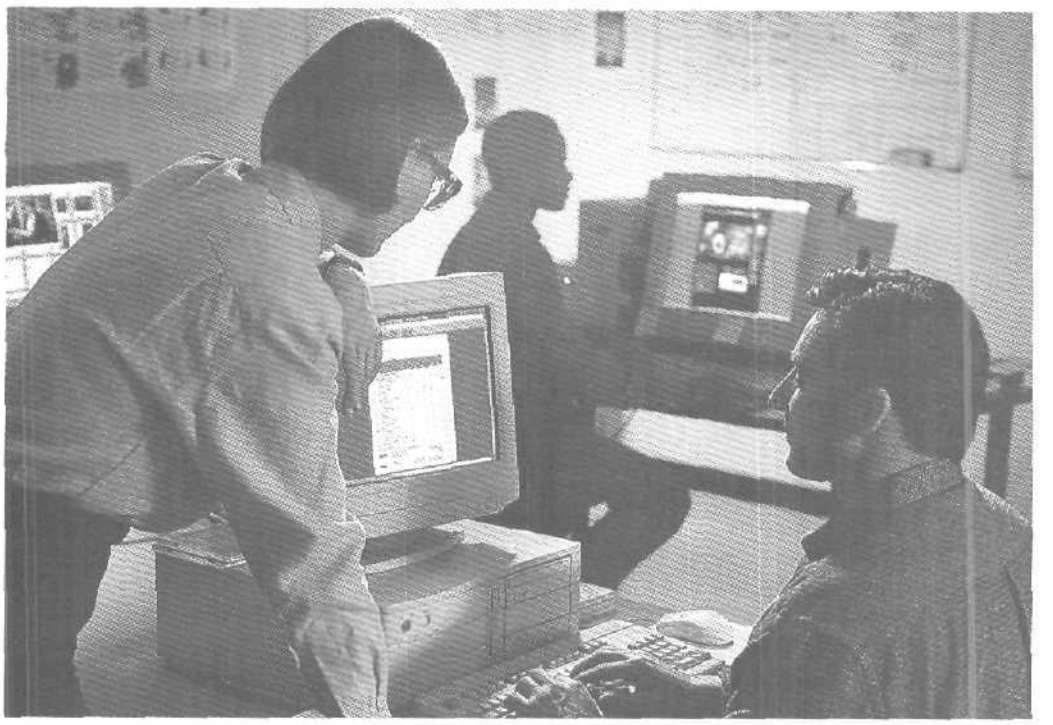
издательство компьютерной литературы
РУССКАЯ РЕДАКЦИЯ

ПРОДАЖА КНИГ
оптом тел.: (095) 142-0571. e-mail: sale@rusedit.ru
интернет-магазин http://www.ITbook.ru. тел.: (095) 145-4519

HARD'n'SOFT

www.hardnsoft.ru

МНОГОГРАННЫЙ
КОМПЬЮТЕРНЫЙ
ЖУРНАЛ



Учебный центр
SoftLine

Ваш курс начинается завтра!

Подготовка сертифицированных инженеров и администраторов Microsoft

Авторизованные и авторские курсы по;

* Windows 2000 / XP
* Sun Solaris 8
* Visual Studio .NET
* Электронной коммерции
* Безопасности информационных систем

и еще более 40 курсов по самым современным компьютерным технологиям,

Дневные и вечерние занятия.

Опытные преподаватели.

Индивидуальные консультации.

softline®
education

Microsoft
CERTIFIED
Technical Education Center

Учебный центр SoftLine

119991 г. Москва, ул. Губкина, д. 8
тел.: (095) 232 00 23
e-mail: educ@softline.ru
http://education.softline.ru

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
softline
ЛИЦЕНЗИРОВАНИЕ • ОБУЧЕНИЕ • КОНСАЛТИНГ
www.softline.ru • 2320023 • info@softline.ru

УЧЕБНЫЙ ЦЕНТР МИКРОИНФОРМ

15 лет в образовании!

Более 150 авторизованных курсов фирм
Microsoft, Novell,
3Com, Huawei, HP/Compaq,
Enterasys Networks, Ortronics,
Avaya, Lexmark, Seiko Epson,
Computer Associates, Allied Telesyn

Курсы для специалистов по информационной безопасности
MIS Training institute

Программы сертификации и подготовки Международного Консорциума по Сертификациям в области Безопасности Информационных Систем (ISC)2

Подготовка пользователей
Школа Web-мастеров

Международная сертификация
Центр тестирования VUE

Адрес 115184, Москва,
ул. Малая Ордынка, 44
Тел.: (095) 953 00 06, факс: (095) 238 83 06
E-mail: educ@microinform.ru