bhv
www.bhv.ru
www.bhv.kiev.ua

Николай Секунов

C#

САМОУЧИТЕЛЬ

Среда разработки приложений Visual Studio.NET

Справочник по языку C#

Создание распределенных приложений

Реализация пользовательского интерфейса

Первый компонентно-ориентированный язык программирования

Николай Секунов

САМОУЧИТЕЛЬ C#

Санкт-Петербург

Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

Книга посвящена первому компонентно-ориентированному языку программирования распределенных приложений C#. Приведены сведения обо всех основных элементах данного языка, начиная с примитивов. Подробно рассматривается предназначенная для разработки приложений среда Microsoft Visual Studio.NET; структура программ на C# и этапы компиляции; объединение компонентов, написанных на различных языках высокого уровня; реализация пользовательского интерфейса; обеспечение безопасности приложений. Изложенная методика создания приложений и многочисленные примеры позволят приобрести устойчивые навыки программирования на языке C#.

Для программистов

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Наталья Таркова*
Редактор	*Алексей Птухин*
Компьютерная верстка	*Виктории Капецкой*
Корректор	*Светлана Симуни*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Секунов Н. Ю.
Самоучитель C#. — СПб.: БХВ-Петербург, 2001. — 576 с.: ил.

ISBN 5-94157-028-7

© Н. Ю. Секунов, 2001
© Оформление, издательство "БХВ-Петербург", 2001

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.06.01.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 46,44.
Тираж 5000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с диапозитивов
в Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9-я линия, 12.

Содержание

Введение **17**

Для кого предназначена эта книга? 17
Структура книги 18
Соглашения, принятые в книге 20

Глава 1. C# и Visual Studio.NET **21**

Объектно-ориентированное программирование 21
Инкапсуляция 23
Наследование 24
Полиморфизм 24
Компонентно-ориентированное программирование 25
Среда Common Language Runtime 26
Компиляторы времени выполнения приложения 28
Система виртуальных объектов 30
Зачем нужен C# 36
Простой 37
Удобный 38
Надежный и безопасный 39
Совместимый 40
Объектно-ориентированный 42
Совместимый с унаследованными технологиями 42
Контролирующий версии 43
Легко распространяемый 43

Глава 2. Создание простейшего приложения **45**

Метод Main 47
Работа с консолью 48
Включение комментариев 50

Глава 3. Первое знакомство с C# **53**

Автоматическое управление памятью 53
Типы переменных 58
Предопределенные типы переменных 58
Массивы 60
Преобразование типов переменных 62
Переменные и формальные параметры 63
Выражения 68
Операторы 69
Последовательности операторов и блоки 69
Метки и оператор goto 69
Условный оператор 70

Оператор-переключатель ... 70
Оператор while ... 71
Оператор do ... 71
Оператор for ... 72
Оператор foreach ... 72
Оператор return ... 73
Операторы throw и try ... 73
Операторы checked и unchecked ... 73
Оператор lock ... 74
Классы ... 74
Константы ... 74
Поля ... 75
Методы ... 76
Свойства ... 77
События ... 78
Операции ... 80
Индексаторы ... 81
Конструкторы объектов класса ... 82
Деструкторы ... 83
Статические конструкторы класса ... 83
Наследование ... 83
Структуры ... 86
Интерфейсы ... 87
Делегаты ... 88
Перечислимые типы ... 90
Пространства имен и сборки ... 90
Работа с версиями ... 93
Атрибуты ... 97
Исключения ... 99

Глава 4. Лексическая структура языка ... 101

Директивы препроцессора ... 101
Определение идентификаторов ... 102
Условная секция ... 104
Управляющие строки препроцессора ... 104
Изменение номера строки ... 105
Выражения препроцессора ... 105
Лексический анализ ... 106
Синтаксический анализ ... 107
Идентификаторы ... 108
Ключевые слова ... 109
Литералы ... 110
Логический литерал ... 110
Целочисленный литерал ... 110
Литерал формата с плавающей точкой ... 111
Символьный литерал ... 112

Строковый литерал 113
Нулевой литерал 114
Операции и знаки препинания 114
Эскейп-последовательность Unicode 114

Глава 5. Пространства имен 115

Члены пространств имен и типов 118
Установка режима доступа 119
Область действия членов 120
Ограничения, налагаемые на режимы доступа 124
Сигнатура и перегрузка членов 125
Область видимости 127
Скрытие имен 129
Использование имен пространств имен и типов 131

Глава 6. Переменные 135

Категории переменных 135
Статические переменные 135
Нестатические переменные 136
Элементы массивов 136
Формальные параметры, передаваемые по значению 136
Формальные параметры, передаваемые по ссылке 137
Возвращаемые значения 137
Локальные переменные 138
Значения по умолчанию 138
Инициализация 138
Ссылки на переменные 142
Типы переменных 142
Типы значений 143
Инициализация переменных типов значений 145
Структуры 146
Простые типы значений 148
Целочисленные типы 150
Типы значений с плавающей точкой 152
Десятичный тип 154
Логический тип 155
Перечислимые типы 155
Типы ссылок 159
Классы 159
Интерфейсы 160
Массивы 160
Типы массивов 161
Инициализация объекта массива 161
Доступ к элементам массива 164
Делегаты 165

Глава 7. Преобразование типов 175

Неявное преобразование типов 175
Тождественное преобразование 175
Неявное преобразование числовых типов 176
Неявное преобразование перечислимых типов 176
Неявное преобразование типов ссылок 177
Упаковка значений 177
Неявное преобразование типов константных выражений 180
Неявные преобразования пользовательских типов 180
Явное преобразование типов 180
Явное преобразование числовых типов 181
Явное преобразование перечислимых типов 183
Явное преобразование типов ссылок 183
Распаковка значения 184
Явные преобразования пользовательских типов 185
Преобразования пользовательских типов 185
Неявное преобразование пользовательских типов 187
Явное преобразование пользовательских типов 189
Преобразования объектов 190
Преобразования классов 190
Преобразования интерфейсов 192
Преобразование структур 193

Глава 8. Выражения 195

Результат выражения 196
Операции 197
Приоритеты и порядок выполнения операций 197
Перегрузка операций 200
Составление списка кандидатов на реализацию перегруженной операции 201
Вызов перегруженных унарных операций 202
Вызов перегруженных бинарных операций 202
Преобразование типов операндов 202
Доступ к члену типа 205
Функциональные члены 206
Список аргументов 209
Поиск реализации 211
Вызов функциональных членов 213
Первичные выражения 215
Литералы 215
Простые имена 216
Выражение в скобках 219
Доступ к члену 219
Использование типов как простых имен 222
Вызов функционального члена 222
Вызов метода 223
Вызов делегата 224

Доступ к элементу 224
Доступ к элементу массива 225
Доступ по индексатору 225
Доступ к символу строки 226
Ключевое слово this 227
Ключевое слово base 227
Постфиксное инкрементирование и декрементирование 228
Операция new 229
Создание объекта 230
Создание массива 231
Создание делегата 233
Операция typeof 236
Операция sizeof 237
Установка и сброс проверки на переполнение 237
Унарные выражения 238
Унарная операция + ...239
Унарная операция 239
Операция логического отрицания 240
Операция побитового отрицания 240
Префиксное инкрементирование и декрементирование 241
Операция преобразования типа 242
Арифметические операции 243
Операция умножения 243
Операция деления 244
Операция деления по модулю 245
Операция сложения 246
Операция вычитания 248
Операции сдвига 249
Операции отношения 250
Целочисленные операции сравнения 251
Операции сравнения для чисел с плавающей запятой 252
Операции сравнения для десятичных чисел 253
Логические операции сравнения 253
Операции сравнения для перечислимых типов 254
Операции сравнения для типов ссылок 254
Операция сравнения строк 255
Операция сравнения делегатов 256
Операция is 256
Операция as 256
Побитовые логические операции 256
Целочисленные побитовые логические операции 257
Побитовые логические операции для перечислимых типов 257
Побитовые логические операции для логического типа 258
Логические операции 258
Условная операция 260
Операции присваивания 261

Простая операция присваивания....261
Составная операция присваивания....263
Константное выражение....264
Логическое выражение....265

Глава 9. Операторы....267

Конечная точка и доступность....267
Блоки....269
Последовательность операторов....269
Пустой оператор....270
Помеченный оператор....270
Оператор объявления....271
Объявление локальных переменных....272
Объявление локальных констант....274
Оператор-выражение....275
Условный оператор....275
Оператор-переключатель....277
Операторы цикла....282
Оператор while....282
Оператор do....283
Оператор for....284
Оператор foreach....285
Операторы перехода....287
Оператор break....288
Оператор continue....289
Оператор goto....289
Оператор return....290
Оператор throw....290
Оператор try....291
Операторы checked и unchecked....294
Оператор lock....295

Глава 10. Классы....297

Реализация класса....297
Модификаторы класса....297
Спецификатор базового класса....299
Члены класса....300
Наследование....301
Модификаторы режима доступа....302
Статические и нестатические члены класса....302
Константы....304
Поля....306
Поля, используемые только для чтения....306
Версии констант и статических полей, используемых только для чтения....307
Инициализация полей....308
Методы....310

Формальные параметры методов 311
Виртуальные методы 315
Перегрузка методов 318
Абстрактные методы 319
Внешние методы 320
Тело метода 320
Свойства 321
Статические и нестатические свойства 322
Процедуры доступа 322
Виртуальные, перегружаемые и абстрактные процедуры доступа 327
События 330
Индексаторы 335
Операции 338
Унарные операции 338
Бинарные операции 339
Операции преобразования типов 339
Конструкторы объектов класса 341
Инициализатор конструктора 342
Инициализация полей класса 342
Конструкторы класса, используемые по умолчанию 345
Закрытые конструкторы классов 345
Деструкторы класса 345
Статические конструкторы класса 347

Глава 11. Интерфейсы 349

Заголовок интерфейса 349
Модификаторы интерфейса 349
Спецификатор базового интерфейса 349
Члены интерфейса 350
Методы 351
Свойства 351
События 351
Индексаторы 351
Доступ к членам интерфейса 352
Полные имена членов интерфейса 354
Реализация интерфейсов 354
Явная реализация членов интерфейса 355
Поиск реализации члена интерфейса 357
Наследование реализации интерфейсов 361
Создание новой реализации интерфейса 363
Абстрактные классы и интерфейсы 364
Использование интерфейсов 365

Глава 12. Атрибуты 369

Классы атрибутов 369
Позиционные и именованные параметры 371

Типы параметров атрибутов 372
Определение атрибутов 372
Объекты атрибутов 373
Компиляция атрибутов 374
Вызов объектов атрибутов 374
Создание пользовательских атрибутов 375
Получение информации об атрибуте 375
Глобальные атрибуты 377
Зарезервированные атрибуты 377
Задания свойств пользовательских атрибутов 378
AttributeUsage 378
Атрибуты условной трансляции 379
conditional 380
Взаимодействие с другими приложениями 384
comimport 384
comsourceinterfaces 385
comvisibility 386
dispid 387
dllimport 388
globalobject 390
guid 390
hasdefaultinterface 391
importedfromcom 392
in 393
interfacetype 393
iscomregisterfunction 394
marshal 395
name 397
noidispatch 398
out 398
predeclared 399
returnshresult 399
structlayout 401
structoffset 403
typelibfunc 404
typelibtype 405
typelibvar 406
Сохранение классов и структур в потоках 406
nonserialized 407
serializable 407
obsolete 408
Используемые перечислимые типы 410
AttributeTargets 410
CallingConvention 411
CharSet 411
COMVisibility 412

ComInterfaceType 412
LayoutKind 412
UnmanagedType 412

Глава 13. Реализация пользовательского интерфейса 415

Работа со строками 415
Форматирование 417
Стандартные строки форматирования 417
Целочисленный десятичный формат 417
Целочисленный шестнадцатеричный формат 418
Формат с фиксированной запятой 418
Экспоненциальный формат 419
Общий формат 420
Формат с выделением тысяч 421
Финансовый формат 421
Модификация формата вывода 422
Форматирование целых чисел 422
Форматирование чисел с плавающей точкой 423
Выделение тысяч 424
Масштабирование 425
Вывод процентов 425
Изменение формата чисел 426
Включение литералов 426
Форматирование объектов 427
Внесение изменений в существующие форматы 428
Преобразование строк в значения 430
Чтение и запись файлов 431
Автоматическое сохранение объектов 432
Работа с потоками 434

Глава 14. Включение отладочной информации 439

Условный вызов методов 439
Классы Debug и Trace 440
Проверка условий 441
Использование ключей для фильтрации выводимой информации 444
Уровни трассировки 445
Использование пользовательских переключателей 447

Глава 15. Обработка исключений 451

Проверка переполнения 452
Операторы try и catch 454
Трансляция исключений 458
Обработка исключения 458
Создание нового исключения 459
Пользовательские классы исключений 461
Оператор finally 463

Глава 16. Обеспечение безопасности.....467

Безопасность доступа к компоненту.....467
Безопасность типов.....468
Права доступа.....468
Стандартные права доступа.....469
Персональные права доступа.....470
Ролевая безопасность.....470

Приложение. Интерфейс пользователя Visual Studio.NET.....473

Первая страница Visual Studio.NET.....475
Панели инструментов.....477
Панель инструментов Standard.....480
Система меню.....482
Меню File.....483
Команда File, New.....483
Команда File, Open.....484
Команда File, Close.....487
Команда File, Add Project.....487
Команда File, Open Solution.....487
Команда File, Close Solution.....488
Команда File, Save (<Ctrl>+<S>).....488
Команда File, Save As.....488
Команда File, Advanced Save Options.....489
Команда File, Save All (<Ctrl>+<Shift>+<S>).....490
Команда File, Source Control.....490
Команда File, Page Setup.....490
Команда File, Print (<Ctrl>+<P>).....490
Команда File, Recent Files.....491
Команда File, Recent Projects.....491
Команда File, Exit.....492
Меню Edit.....492
Команда Edit, Undo (<Ctrl>+<Z>).....492
Команда Edit, Redo (<Ctrl>+<Y>).....493
Команда Edit, Cut (<Ctrl>+<X>).....494
Команда Edit, Copy (<Ctrl>+<C>).....494
Команда Edit, Paste (<Ctrl>+<V>).....494
Команда Edit, Delete (<Del>).....494
Команда Edit, Select All (<Ctrl>+<A>).....494
Команда Edit, Find and Replace.....495
Команда Edit, Go To Line (<Ctrl>+<G>).....503
Команда Edit, Insert File As Text.....504
Команда Edit, Advanced.....504
Команда Edit, Bookmarks.....506
Команда Edit, Outlining.....507
Команда Edit, IntelliSense.....510
Меню View.....513

Команда View, Code (<F7>) 513
Команда View, Open 513
Команда View, Open With 513
Команда View, Solution Explorer (<Ctrl>+<Alt>+<L>)....514
Команда View, Class View (<Ctrl>+<Shift>+<C>)....515
Команда View, Server Explorer (<Ctrl>+<Alt>+<S>)....515
Команда View, Resource View (<Ctrl>+<Shift>+<E>) 515
Команда View, Properties Window (<F4>) 515
Команда View, Toolbox (<Ctrl>+<Alt>+<X>)....516
Команда View, Web Browser (<Ctrl>+<Alt>+<R>)....517
Команда View, Other Windows....517
Команда View, Show Tasks 517
Команда View, Toolbars 519
Команда View, Full Screen (<Shift>+<Alt>+<Enter>)....519
Команда View, Server Explorer View....520
Команда View, Navigate Forward 521
Команда View, Property Pages....521
Меню Project....522
Команда Project, Add Windows Form 523
Команда Project, Add Inherited Form 523
Команда Project, Add User Control....523
Команда Project, Add Inherited Control 523
Команда Project, Add Component....524
Команда Project, Add Class 524
Команда Project, Add Method....524
Команда Project, Add Property....525
Команда Project, Add Field 526
Команда Project, Add Indexer 526
Команда Project, Add New Item (<Ctrl>+<Shift>+<A>)....526
Команда Project, Add Existing Item (<Ctrl>+<Alt>+<A>) 527
Команда Project, Exclude From Project 528
Команда Project, Show All Files 528
Команда Project, Add Reference....528
Команда Project, Add Web Reference 529
Команда Project, Set as StartUp Project....529
Меню Build....529
Команда Build, Build (<Ctrl>+<Shift>+<B>)....530
Команда Build, Rebuild....530
Команда Build, Deploy 531
Команда Build, Batch Build 531
Команда Build, Configuration Manager....531
Меню Debug....532
Команда Debug, Windows....533
Команда Debug, Start (<F5>)....533
Команда Debug, Break (<Ctrl>+<Alt>+<Break>)....534
Команда Debug, Stop Debugging (<Shift>+<F5>)....534

Команда Debug, Detach All 534
Команда Debug, Restart (<Ctrl>+<Shift>+<F5>) 534
Команда Debug, Apply Code Changes (<Alt>+<F10>) 534
Команда Debug, Start Without Debugging (<Ctrl>+<F5>) 534
Команда Debug, Processes 534
Команда Debug, Step Into (<F11>) 535
Команда Debug, Step Over (<F10>) 535
Команда Debug, Step Out (<Shift>+<F11>) 536
Команда Debug, Step By 536
Команда Debug, Set Next Statement 537
Команда Debug, Run to Cursor (<Ctrl>+<F10>) 537
Команда Debug, Go To Disassembly 537
Команда Debug, QuickWatch (<Ctrl>+<Alt>+<Q>) 538
Команда Debug, New Breakpoint (<Alt>+<F9>+<N>) 538
Команда Debug, Clear All Breakpoints (<Ctrl>+<Shift>+<F9>) 538
Команда Debug, Save As CrashDump 538
Команда Debug, Disable Breakpoint (<Ctrl>+<F9>) 540
Меню Tools 540
Команда Tools, Connect to Database 540
Команда Tools, Connect to Server 541
Команда Tools, Customize Toolbox 542
Команда Tools, Add-in Manager 542
Команда Tools, Build Comment Web Pages 542
Команда Tools, Macros 544
Команда Tools, Trace Tool 544
Команда Tools, OLE/COM Object Viewer 546
Команда Tools, Spy+ 546
Команда Tools, External Tools 546
Команда Tools, Customize 547
Команда Tools, Options 548
Меню Window 549
Команда Window, New Window 549
Команда Window, Split 549
Команда Window, Dockable 551
Команда Window, Hide 551
Команда Window, Floating 551
Команда Window, Auto Hide 551
Команда Window, Auto Hide All 552
Команда Window, New Horizontal Tab Group 552
Команда Window, New Vertical Tab Group 554
Команда Window, Move to Next Tab Group 555
Команда Window, Move to Previous Tab Group 555
Команда Window, Close All Documents 555
Список открытых окон 555
Команда Window, Windows 555

Меню Help 556
Команда Help, Dynamic Help (<Ctrl>+<F1>)....556
Команда Help, Contents (<Ctrl>+<Alt>+<F1>)....556
Команда Help, Index (<Ctrl>+<Alt>+<F2>)....556
Команда Help, Search (<Ctrl>+<Alt>+<F3>) 559
Команда Help, Index results (<Shift>+<Alt>+<F2>)....559
Команда Help, Search results (<Shift>+<Alt>+<F3>)....560
Команда Help, Edit Filters....560
Остальные команды меню....562
Окна Visual Studio.NET 562
Окно Solution Explorer 562
Окно Class View 562
Окно Properties 564
Окно Watch 564
Окно Breakpoints....564

Предметный указатель 567

Введение

В процессе конкурентной борьбы перед корпорацией Microsoft встала задача приостановить победное шествие языка программирования Java, разработанного корпорацией Sun. Попытки расколоть стандарт этого языка изнутри путем введения в Visual Java элементов, ускоряющих его работу, но делающих его несовместимыми со всеми платформами, за исключением Windows, привели к судебному разбирательству, которое закончилось не в пользу Microsoft. Ответом на это стала разработка языка программирования C#, решающего те же задачи, но являющегося лицензионно чистым, хотя в нем в большой степени учтен опыт разработки языка Visual Java.

Язык C# представляет собой язык программирования более высокого уровня, чем язык C++ и может рассматриваться как его дальнейшее усовершенствование. Изначально язык C разрабатывался как инструментальный язык операционной системы UNIX. Поэтому перед его транслятором ставилась задача создания машинных кодов, не уступающих по эффективности машинным кодам, созданным языком ассемблера. Такие жесткие требования наложили свой отпечаток на синтаксис языка. Для расширения возможностей языка C на его основе был создан объектно-ориентированный язык C++, который иногда называли "C с классами". C++ идеально подходит для создания приложений на одном компьютере. Однако в настоящее время все большее распространение получают распределенные приложения. При их создании программист имеет дело с обширными библиотеками готовых программ, которые нужно только объединить в одно приложение. В этом случае требования эффективности создаваемого машинного кода отступают на второй план, поскольку эффективность данных приложений в основном определяется эффективностью используемых библиотек, а на первый план выступает обеспечение надежности взаимодействия объединяемых компонентов приложения, которые могут быть созданы разработчиками, не имеющими друг о друге ни малейшего представления. Язык C# как раз и является языком программирования распределенных приложений.

Для кого предназначена эта книга?

Язык программирования C# является абсолютно новым и его стандарт еще не устоялся, однако, уже сейчас к нему проявляется определенный интерес со стороны тех программистов, которые хотели бы оценить возможности его

использования для реализации своих проектов. Пока что единственной средой программирования, в которой реализован язык C#, является среда Microsoft Visual Studio 7.0, известная, так же как Microsoft Visual Studio.NET, поэтому в данной книге рассмотрен не только сам язык, но и его поддержка со стороны среды программирования.

В приложении приведено описание интегрированной среды разработки Microsoft Visual Studio.NET, в которой тестировались все приведенные в книге примеры. Кроме того, там же дается краткое описание среды исполнения приложений, написанных на языке C#, что позволяет лучше понять его особенности и логику построения. В главе 2 изложена методика построения консольных приложений C#, используя которые можно получить навык работы практически со всеми конструкциями описываемого языка (по крайней мере, все приведенные в данной книге примеры представляют собой консольные приложения или же их фрагменты).

Структура книги

Как и в большинстве книг по C и C++, в книге, предлагаемой вашему вниманию, сначала дано краткое описание языка программирования C#, а затем подробно рассмотрены некоторые его аспекты. В приведённом ниже списке глав приводится краткое их описание, позволяющее пользователю лучше ориентироваться в структуре книги.

- Глава 1. C# и Visual Studio.NET.

  В главе рассмотрены концепции объектно-ориентированного и компонентно-ориентированного программирования. Рассмотрена так же реализация концепции компонентно-ориентированного программирования в среде .NET и роль языка C# в этой реализации. Здесь же изложены основные особенности данного языка программирования.

- Глава 2. Создание простейшего приложения.

  В главе описана процедура создания простейшего консольного приложения в среде Visual Studio.NET. Здесь также содержится описание структуры консольного приложения и основных его компонентов, что позволит читателю подобные приложения самостоятельно создавать и модифицировать.

- Глава 3. Первое знакомство с C#.

  В главе приведено краткое описание языка программирования C#. Эту главу можно рассматривать как краткий конспект глав последующих. Здесь же подробно обсуждаются некоторые вопросы, вынесение которых в отдельную главу, по мнению автора, было бы нецелесообразно.

- Глава 4. Лексическая структура языка.

  В главе рассмотрена структура языка и этапы его компиляции, содержится описание директив препроцессора и определены понятия идентификатора, ключевого слова, литерала, операции и знака препинания.

- Глава 5. Пространства имен.

  В главе рассмотрена структура программ C#, определяются режимы доступа и область действия членов пространств имен и типов. Здесь же рассмотрено скрытие имен и другие аналогичные вопросы.

- Глава 6. Переменные.

  Глава посвящена переменным, используемым в языке C#, описанию стандартных операций данного языка, а также способам обращения к его элементам в выражениях.

- Глава 7. Преобразование типов.

  В главе идет речь об операциях преобразования типов в C#, дан полный перечень допустимых явных и неявных преобразований типов, а также подробно описан алгоритм каждого из этих типов преобразований.

- Глава 8. Выражения.

  В главе рассмотрены выражения, используемые в языке C#, стандартные операции и их перегрузка, описана работа с функциональными членами типов, поиск требуемой реализации и другие связанные с этим вопросы.

- Глава 9. Операторы.

  Глава посвящена операторам языка C#.

- Глава 10. Классы.

  В главе рассматривается работа с классами языка C#, наследование классов, создание статических и виртуальных членов класса, скрытие членов класса, а также дано описание всех членов класса.

- Глава 11. Интерфейсы.

  В главе описана работа с интерфейсами языка C#. Рассмотрено наследование интерфейсов, их реализация в классах и структурах, а также дано описание всех членов интерфейса.

- Глава 12. Атрибуты.

  Глава посвящена атрибутам языка C#. Атрибуты используются для связывания с программными единицами C# некоторой информации, которая может быть получена в процессе выполнения приложения.

- Глава 13. Реализация пользовательского интерфейса.

  В главе рассматриваются методы работы со строками, форматирование выводимой информации, модификация формата, а также работа с файлами и потоками.

- Глава 14. Включение отладочной информации.

  В главе идет речь о классах и атрибутах, позволяющих включать отладочные функции не только в отладочную, но и в распространяемую версию приложения.

- Глава 15. Обработка исключений.

  В главе описываются принципы обработки исключений в языке C#. В этой же главе рассмотрен способ управления проверкой переполнения при выполнении операций над целыми числами.

- Глава 16. Обеспечение безопасности.

  Глава посвящена методам, используемым в среде .NET для обеспечения безопасности использования компонентов, полученных из ненадежных источников, например, из Internet.

Соглашения, принятые в книге

В книге автором использовалось специальное форматирование для выделения некоторых текстов или фрагментов текста. Ниже приведены основные принципы используемого форматирования:

- исходные тексты фрагментов программ, представляющие собой одну или более строк текста, выделяются специальным шрифтом, как это показано ниже:

```
class Some_Class
{
        private static bool   isEmpty;       // Статическое поле
        public string         Name;          // Нестатическое поле
}
```

- имя класса, функции, имя типа переменной или фрагмент текста длиной менее строки выделяются специальным шрифтом. Например: `int`, `System`, `name` и т. д.;
- имя клавиши заключается в угловые скобки (<>) и выводится жирным шрифтом. Например: <Ctrl>, <F5> и т. д.;
- при первом появлении нового термина он выделяется курсивом. Например: *новый термин*;
- ссылка на другую главу содержит только ее номер. Например: *См. главу 5.*

Глава 1

C# и Visual Studio.NET

На протяжении всей истории программирования особое внимание уделялось выработке единого, стандартного подхода к проектированию и реализации компьютерных программ. Потребность в таком подходе обуславливалась прежде всего усложнением задач, решаемых программистами, и необходимостью максимально уменьшить число ошибок в программах еще на этапе их разработки. Первые попытки выработки единого подхода к программированию связаны с так называемым структурным программированием.

Приверженцы структурного подхода к программированию пытались разработать некоторые универсальные методы проектирования систем, организации данных и проектирования программ. Однако такой подход не получил широкого распространения, поскольку не содержал никаких революционных идей и представлял собой, по сути, набор благих пожеланий и полезных советов. Революция в разработке программного обеспечения наступила с появлением концепции объектно-ориентированного программирования.

Объектно-ориентированное программирование

Концепция объектно-ориентированного программирования позволяет рассматривать программу как набор модулей, каждый из которых выполняет отведенную ему функцию. Если вам необходимо отправить письмо, то вы не идете на вокзал, не отыскиваете человека, едущего в тот же город и не просите его передать письмо по определенному адресу. Вы просто опускаете письмо в почтовый ящик и надеетесь, что оно вовремя дойдет до адресата. Точно также, если вам необходимо вызвать на экран окно, то вам совсем не обязательно разбираться в том, каким образом отображается его рамка, а тем более, каким образом производится обработка поступающих в него и отправляемых им сообщений. Тем более что, если вы попытаетесь это сделать, то вы что-нибудь упустите (а это естественно при отсутствии у вас полного описания системы) и, скорее всего, вызовете крах всей системы.

Однако концепция объектно-ориентированного программирования (ООП) не ограничивается только предоставлением пользователю удобных библио-

тек для решения сложных задач. Эти вопросы могут быть решены и без использования данной концепции, например, в рамках структурного программирования. Программные объекты были разработаны для облегчения процесса программирования и приближения структуры программ к привычным для человека формам.

Можно смело сказать, что именно способность обобщать, классифицировать и генерировать абстракции явилась причиной превращения обезьяны в человека. Без этой способности невозможно было бы создание языка общения, поскольку каждое имя существительное является по сути своей классом объектов, обладающих некоторым набором атрибутов или отличительных черт. С другой стороны, каждый подобный класс является базовым для других классов, формируемых, например, путем добавления к этому существительному конкретизирующего его прилагательного. Например, "белый дом".

В основе концепции объектно-ориентированного программирования лежат три основные принципа:

1. Инкапсуляция:

   Совмещение структур данных с функциями (методами), предназначенными для манипулирования этими данными. Инкапсуляция достигается путем введения нового механизма структурирования и типизации данных — класса.

2. Наследование:

   Создание новых, производных классов, которые наследуют данные и функции от одного или нескольких ранее определенных базовых классов. При этом возможно переопределение или добавление новых данных и методов. В результате создается иерархия классов.

3. Полиморфизм:

   Присвоение методу единого имени или идентификатора в рамках иерархии классов, чтобы любой класс в иерархии имел возможность по-своему выполнять связанные с этим методом действия.

Одновременно с появлением и детализацией концепции ООП появились и основанные на ней языки программирования. Одним из первых языков стал алгоритмический язык Modula 2. Язык программирования Turbo Pascal, разработанный фирмой Borland, начиная с версии 5.5, превратился в объектно-ориентированный. Но наиболее последовательное воплощение концепция объектно-ориентированного программирования нашла в алгоритмическом языке C++, разработанном сотрудником Bell Laboratories AT&T Брайаном Страуструпом на базе языка программирования C как средство моделирования больших телефонных коммутационных систем.

Рассмотрим основополагающие принципы ООП подробнее.

Инкапсуляция

Одним из главных отличий стандартного структурного программирования от объектно-ориентированного программирования является инкапсуляция. Совмещение структур данных с методами, предназначенными для манипулирования этими данными, позволяет вам сосредоточить в одном месте всю информацию, касающуюся определенных данных и методов их обработки, и тем самым облегчить доступ к этим методам и данным из любой точки программы.

Действительно, представьте себе, что все данные в программе имеют глобальную область видимости (что неизбежно при тесном взаимодействии всех частей программы). При этом вам придется для всех данных и для каждого метода использовать уникальный идентификатор, причем этот идентификатор должен быть информативным, то есть длинным. Программы с такими идентификаторами станут практически нечитаемыми, особенно на экранах дисплеев. Вам придется постоянно следить, чтобы новые имена не пересекались с уже используемыми, чтобы в каждой функции использовались именно те данные, для которых она была написана. Кроме того, вам придется отказаться от таких элегантных возможностей, как перегрузка операций и создание собственных типов данных, позволяющих автоматически проверять корректность выполняемых над ними операций.

Кроме того, инкапсуляция позволяет вам скрыть данные и функции, используемые исключительно членами данного класса или членами производного от него класса. Это упрощает описание класса для его использования внешними функциями и повышает надежность его работы, поскольку критически важные данные могут быть, таким образом, защищены от несанкционированного доступа.

Хорошим тоном считается скрытие всех данных, используемых данным классом, и обеспечение доступа к ним только посредством соответствующих методов класса. Это прежде всего устраняет все вопросы, связанные с областью видимости переменных, и позволяет использовать в каждом классе простые и информативные идентификаторы, не заботясь о том, что подобные идентификаторы уже используются в других классах.

Обеспечение доступа к данным исключительно через методы класса дает возможность избегать потенциально очень опасных ситуаций. Предположим, что в процессе разработки какого-либо проекта одному из разработчиков была поставлена задача написать обработчик для некоторого массива, содержащего указатели на символьные строки. На каком-то этапе этот разработчик решил заменить этот массив связанным списком и соответствующим образом переписал методы обращения к нему.

Если он оформил этот массив в виде класса и обеспечил доступ к массиву исключительно через методы этого класса, то другие разработчики ни при каких обстоятельствах не должны почувствовать последствия этой замены

(разве что она может отразиться на скорости выполнения приложения). Но если к этому массиву был обеспечен прямой доступ, то никто не может поручиться, что другой программист, работающий с теми же данными, не захочет воспользоваться собственными, более эффективными с его точки зрения методами работы с массивом, которого уже нет.

Наследование

Принцип наследования в объектно-ориентированном программировании позволяет создать класс, включающий в себя все данные и методы некоторого другого класса, и, кроме того, содержащий свои, присущие только ему данные и методы обработки. Такой подход позволяет строить целые иерархии классов, каждый следующий уровень которых отличается все большей детализацией по сравнению с предшествующими ему уровнями. Достоинства такого подхода наиболее полно проявляются при программировании достаточно сложных объектов, к которым по праву относится сама операционная система Windows.

При всем своем различии окно представления и диалоговое окно являются окнами Windows и, как таковые, имеют много общего. Представьте себе, во что превратилось бы описание класса диалогового окна, если бы в нем непосредственно были бы описаны все обрабатываемые им данные и все методы обработки, учитывая тот факт, что каждое текстовое поле и каждый переключатель по сути являются самостоятельным и полноправным окном Windows. Не говоря уже о том, сколько места займет описание каждого класса и сколько в них будет повторяющихся и практически аналогичных по тексту функций.

Наследование позволяет описать сначала общий класс окна Windows, на его основе описать общий класс диалогового окна Windows, а уже на основе этого класса сформировать класс конкретного диалогового окна, используемого для реализации конкретного пользовательского интерфейса.

Полиморфизм

Последней отличительной чертой объектно-ориентированного программирования является полиморфизм. Использование полиморфизма позволяет существенно расширить набор функций в базовом классе, что, в свою очередь, приводит к существенному сокращению текста программы и улучшению ее читабельности.

Греческое слово "полиморфизм" было заимствовано из биологии, где оно означает наличие в пределах одного вида резко отличающихся по облику особей. В C++ полиморфизм реализуется с помощью так называемых виртуальных функций, которые позволяют в пределах иерархии классов использовать одно имя метода для выполнения одной и той же операции над различными типами данных. Решение, какая именно версия данного метода

будет использована в каждом конкретном случае, принимается на этапе исполнения программы и носит название позднего связывания.

Суть полиморфизма может быть наглядно проиллюстрирована на следующем примере. Предположим, что вы решили создать иерархию классов для манипулирования отображаемыми на экране графическими объектами. Вы создали базовый класс, в который включили среди прочих функцию перемещения графических объектов. Казалось бы, данную функцию очень просто реализовать: необходимо только определить координаты нового положения объекта, уничтожить его изображение на старом месте и нарисовать на новом.

Однако при ближайшем рассмотрении перед вами встает, казалось бы, неразрешимая задача: функции отображения конкретных графических объектов задаются в производных классах, а обращение к ним предполагается производить в базовом классе. Конечно, у данной проблемы существует простое решение: перенести функцию перемещения объекта из базового в производный класс и для каждого класса написать свой вариант функции. Однако если следовать такой логике, в базовом классе практически не останется методов обработки, а все преимущества наследования будут практически сведены к нулю.

Поэтому, как уже упоминалось выше, в рамках ООП был разработан аппарат *виртуальных функций*. Если в производном и базовом классах одна и та же функция объявлена виртуальной (для этого функции должны иметь не только одно и то же имя, но и один и тот же набор формальных параметров и одинаковый тип возвращаемой величины), то при обращении к этой функции из любого метода, унаследованного данным классом от своих базовых классов, будет вызываться функция, определенная в данном классе, а не функция соответствующего базового класса, как это имело бы место в противном случае.

Если виртуальные функции имеют различный набор формальных параметров, то они рассматриваются как разные функции. Различный тип возвращаемой величины у виртуальных функций с одинаковым именем и набором формальных параметров рассматривается как ошибка.

Компонентно-ориентированное программирование

Объектно-ориентированное программирование позволило перейти на новый этап сложности разрабатываемых приложений, поскольку используемые в нем объекты (классы, структуры, интерфейс и т. д.) могли разрабатываться и тестироваться раздельно, что существенно облегчало их разработку и повышало их надежность. Отлаженные объекты, объединенные в библиотеки динамической компоновки, могли быть впоследствии использованы другими программистами для решения своих задач.

Однако развитие сетевых приложений (особенно это касается Сети, как часто называют Internet) поставило вопрос о необходимости перехода на следующий уровень разработки программного обеспечения. Теперь программной единицей становится не объект, а компонент, под которым можно понимать отдельное приложение, распространяемое по сети. Компьютер, на который поступило данное приложение, должен запустить его на исполнение и предоставить ему все необходимые ресурсы. Поскольку эти приложения являются довольно самостоятельными, работающее с ними серверное приложение может не отличаться особой сложностью, но перед ним во всей своей полноте встает проблема обеспечения безопасности.

Когда мы говорим о безопасности, имеются в виду не вирусы и атаки троянцев, хотя и от них нужно защищаться, а тот простой факт, что серверное приложение должно координировать работу приложений, разработчики которых, скорее всего, даже не подозревали о существовании друг друга и, конечно же, не проверяли разрабатываемые ими приложения на совместимость. Кроме того, никто не может гарантировать, что для написания всех координируемых приложений использовался один и тот же язык программирования. Поэтому следует изначально полагать, что в данной среде должны работать приложения, написанные на разных языках программирования.

Поставленные задачи предъявляют довольно-таки специфические требования к языку программирования, на котором будет написано приложение сервера. В свое время корпорация Microsoft попыталась решить эту проблему, создав технологию COM (Component Object Module, Модель Многокомпонентных Объектов). Данная технология оказалась вещью в себе, поскольку ее использование в широко распространенных языках программирования (например, Visual C++) было сопряжено с определенными трудностями. Не говоря уже о том, что в других не менее распространенных языках программирования, таких как Visual Basic, некоторые ее возможности были вообще недоступны.

Убедившись, что технология COM не решает поставленных перед нею задач, Microsoft создала новую технологию .NET Runtime, основным языком которой и является C#. В этой технологии один способ описания программного кода (метаданные), одна среда выполнения (Common Language Runtime) и одна базовая библиотека (Frameworks).

Среда Common Language Runtime

Технология .NET Runtime реализована в среде Common Language Runtime (CLR). Среду CLR не стоит путать со средой разработки Visual Studio, которую можно рассматривать как надстройку над CLR. CLR лучше интерпретировать как некоторый препроцессор, преобразующий команды некоторого промежуточного языка *Intermediate Language* (IL) в команды процессора. CLR не только предоставляет средства, облегчающие программирование, но и обеспечивает выполнение программного кода

Использование CLR для разработки приложения обеспечивает:

- межъязыковую интеграцию (используется Common Language Specification);
- автоматическое управление памятью (сборка мусора);
- независимую от языка обработку исключений;
- контроль типов данных;
- контроль версий программ.

Большинство из этих возможностей CLR реализовано за счет использования метаданных. Метаданные представляют собой объединение программы и данных в файле формата Portable Executable (PE). Приложение, создаваемое с использованием CLR, называется контролируемым приложением. Это приложение содержит метаданные, используемые средой CLR для проверки соответствия имеющихся версий компонентов требованиям данного приложения. Объединение программного кода и данных позволяет снять многие проблемы с системным реестром.

Одной из приятных особенностей CLR является автоматическое управление памятью и сборка мусора. Среда сама выявляет объекты, на которые нет ни одной ссылки, и уничтожает их. Единственный недостаток такого подхода — невозможность ручного уничтожения объектов, что приводит к некоторому перерасходу оперативной памяти. Однако если задуматься, немногие злоупотребляли возможностью ручного удаления объектов при программировании на C++. Как правило, память выделялась в конструкторе класса и освобождалась в его деструкторе.

Как уже говорилось выше, создаваемый компилятором C# контролируемый программный код представляет собой не набор инструкций процессора, а код на некотором промежуточном языке Intermediate Language. Преимущество промежуточного кода состоит в его независимости от типа процессора. Для преобразования этого кода в команды процессора на компьютере должен быть установлен компилятор промежуточного языка.

Программа на промежуточном языке содержит не только сам программный код, но и метаданные, предоставляющие среде выполнения информацию обо всех используемых типах, сигнатурах всех элементов этих типов и другую информацию, которая в приложениях COM помещалась в библиотеки типов и системный реестр. Таким образом, программа и вся необходимая ей информация размещена в одном файле, что существенно облегчает установку распространяемого в нем приложения в системе, поскольку исключает операции регистрации в системном реестре и поиск соответствующей библиотеки типов. Формат файлов, создаваемых средой Common Language Runtime, является расширением формата Portable Executable, используемого хранения исполняемых файлов (.exe) и библиотек динамической компоновки (.dll).

Чтобы лучше понять, что собой представляет IL-код, необходимо рассмотреть его примитивы, то есть элементы, на которые он разлагает программу на языке C#. Ниже приведен далеко не полный список этих примитивов:

- арифметические и логические операции;
- условные операции;
- операции выделения и освобождения памяти;
- работа со стеком;
- передача аргументов;
- объектная модель;
- создание объектов;
- работа с исключениями;
- массивы;
- типизированные адреса.

Компиляторы времени выполнения приложения

Как уже говорилось выше, компиляторы C# и других языков программирования в Visual Studio.NET создают контролируемый код, представляющий собой программный код на языке Intermediate Language. Поскольку этот язык существенно отличается от языка команд процессора, перед тем как этот код будет исполнен, его следует откомпилировать. Этот процесс происходит в процессе выполнения приложения, поэтому используемые компиляторы носят название компиляторов времени выполнения приложения (Just-in-Time Compiler или JIT-компиляторы).

Компиляция в процессе выполнения приложения позволяет существенно сэкономить память, поскольку компоненты, как правило, представляют собой достаточно объемные и многофункциональные приложения, большинство функций которых не используется при решении конкретной задачи, а программа на языке высокого уровня обычно занимает намного меньше места, чем исполняемый код. JIT-компиляторы достаточно эффективны, поскольку при создании промежуточного языка основное внимание уделялось именно эффективности работы компилятора, а не удобству работы с ним разработчика, как это имеет место в традиционных языках программирования высокого уровня. Наличие промежуточного этапа компиляции не должно вызывать у разработчиков особых опасений. Если опасения все-таки возникнут, стоит вспомнить о том, что этот этап преобразования программного кода, скорее всего, не последний, поскольку все современные процессоры преобразуют команды формата x86 в свои внутренние команды, что только способствует повышению их быстродействия.

Работа с компонентом, использующим промежуточный язык, происходит следующим образом: при загрузке компонента для каждого метода каждого из его типов создается заглушка, передающая управление JIT-компилятору, преобразующему IL-код данного метода в набор команд процессора. При вызове метода управление передается этой заглушке. При первом обращении к заглушке после получения машинного кода в ней заменяется указатель, по которому производится передача управления. Теперь он указывает на скомпилированный машинный код данного метода.

В среде CLR для Windows используется не один, а целых три JIT-компилятора. Ниже приведено их краткое описание:

- JIT — компилятор, используемый в CLR по умолчанию. Этот компилятор предварительно анализирует поток выполнения, в который будет включен исходный код, и создает оптимизированный контролируемый машинный код. Этот компилятор использует большой объем памяти, требователен к другим системным ресурсам и его работа занимает достаточно много времени. Этот компилятор может работать с полным набором команд Intermediate Language.
- EconoJIT — представляет собой упрощенную версию JIT-компилятора. Этот компилятор поддерживает кэширование создаваемого машинного кода, но оптимальность этого кода оставляет желать лучшего. EconoJIT рекомендуется использовать при недостаточном объеме оперативной памяти, поскольку он позволяет отбрасывать неиспользуемый скомпилированный код и таким образом уменьшать объем программы. Следует отметить также и высокое быстродействие данного компилятора.
- PreJIT — используется для преобразования IL-кода в машинный код до запуска приложения. Такая стратегия может использоваться в том случае, если компонент имеет малый размер и существует большая вероятность того, что большинство его методов будут использованы. Как можно догадаться, предварительная компиляция существенно сокращает время загрузки и старта приложения, хотя и создает некоторые проблемы с версиями компонент, поскольку замена самой компоненты не означает автоматической замены ее откомпилированного кода, как это имеет место при традиционном подходе. Этот компилятор запускается, например, при установке компонент NGWS.

Выбор JIT-компилятора и настройка параметров его работы производится утилитой JIT Compiler Manager. Эта утилита содержится в файле jitman.exe, расположенном в поддиректории /bin папки, в которую установлен пакет Microsoft NGWS SDK. Эта программа работает в фоновом режиме, поэтому при ее запуске не выводится никакого окна, а ее значок помещается в область системных программ панели задач (рядом с часами). Для ее настройки дважды щелкните левой кнопкой мыши на этом значке. Появится диалоговое окно **JIT Compiler Manager**, изображенное на рис. 1.1.

Рис.1.1. Диалоговое окно **JIT Compiler Manager**

Рассмотрим подробнее его элементы управления:

- **Use EconoJIT only** — как уже говорилось, в CLR по умолчанию используется компилятор JIT. При установке этого флажка будет использоваться компилятор EconoJIT.
- **Optimize For Size** — создаваемый машинный код оптимизируется по размеру, а не по быстродействию, как это имеет место по умолчанию.
- **Limit Size of Code Cache** — ограничивает размер кэша объемом, установленным в текстовом поле **Max Size of Cache (bytes)**. По умолчанию этот флажок не установлен, что позволяет использовать под кэш весь доступный объем оперативной памяти. В кэше размещается скомпилированный код, поэтому необходимо, чтобы заданный объем кэша превышал максимальный размер кода метода.
- **Enable Concurrent GC** — помещает процедуру сборки мусора (garbage collection) в отдельный поток. По умолчанию эта процедура выполняется в потоке пользовательского кода, что в некоторых случаях может замедлить реакцию приложения на внешние воздействия. Процедура параллельной сборки мусора работает медленнее, чем та же процедура, помещенная в пользовательский поток. Кроме того, выделение этой процедуры в отдельный поток возможно, по имеющимся сведениям, только в Windows 2000.
- **Max Code Pitch Overhead (%)** — текстовое поле, в котором можно указать максимальную долю времени, которое должно расходоваться на компиляцию. При превышении этого порога автоматически увеличивается размер кэша кода. Это текстовое поле используется только компилятором EconoJIT, хотя по непонятным причинам остается доступным и при сбросе соответствующего флажка.

Система виртуальных объектов

Правила объявления и использования типов в среде CLR определяются системой виртуальных объектов (NGWS Virtual Object System или VOS). Использование системы VOS упрощает межъязыковую интеграцию и обеспечивает безопасность типов без существенного снижения производительности создаваемого приложения.

В языке C# используются четыре основные составляющие VOS:

1. Система типов — предназначена для унификации типов данных, используемых различными языками программирования.
2. Метаданные — содержат описание и ссылки на типы, определенные в данной системе. Формат метаданных определяется только системой VOS. Поэтому метаданные могут использоваться для обеспечения взаимодействия между инструментами и виртуальной средой выполнения.
3. Common Language Specification — спецификация единого языка.
4. Virtual Execution System — уже упомянутая выше виртуальная среда выполнения. Эта среда представляет собой реализацию VOS и отвечает за загрузку и выполнение программ, использующих среду CLR.

Рассмотрим подробнее каждую из перечисленных выше составляющих VOS.

Система типов VOS

Система типов VOS представляет собой универсальную систему типов, к элементам которой может быть преобразован любой тип из широкого диапазона языков программирования. Этот диапазон включает в себя не только объектно-ориентированные, но и процедурные языки программирования, поэтому использование такой системы существенно облегчает межъязыковое взаимодействие.

В процессе развития компьютерных технологий и сопровождавшего его процесса появления и развития языков программирования возникло множество сходных типов, различающихся своими параметрами. Наиболее ярким примером можно считать целочисленный тип. При работе с 16-разрядными компьютерами целочисленный тип в большинстве языков программирования имел длину 16 разрядов. Для упрощения переноса программ на 32-разрядные компьютеры длина базового целочисленного типа большинства языков программирования (например, тип `int` в C++) стала составлять 32 разряда. Другие языки программирования такие, как Visual Basic, сохранили 16-разрядные целые числа в качестве своего основного целочисленного типа, что приводит к необходимости согласования типов. В качестве других примеров могут быть рассмотрены различные форматы даты и времени, а также форматы записей баз данных.

Необходимо учитывать и то, что отсутствие жестких стандартов на языки программирования позволило отдельным разработчикам компиляторов языков вносить в них отдельные усовершенствования, приводящие к невозможности совместного использования компонентов, разработанных в различных средах программирования (такой подход является центральным в маркетинговой политике Microsoft), что существенно ограничивает повторное использование программного кода. Эта проблема частично решена в COM за счет использования двоичного стандарта интерфейса.

Основным препятствием при использовании распределенных приложений является различие объектных моделей в разных языках программирования. Эти различия встречаются повсеместно: в формате событий, в свойствах объектов, во времени их существования и во многих других вопросах. Использование унифицированной системы типов VOS позволяет решить все перечисленные выше проблемы.

В системе типов VOS различают два основных типа элементов:

1. Тип значения определяет объем оперативной памяти, необходимый для хранения некоторой информации, и набор стандартных операций, которые могут быть над ней произведены.
2. Тип объекта отличается от типа значения тем, что в нем определен тип нестандартных операций и преобразований, которые могут быть произведены над хранящейся в нем информацией.

Таким образом, тип значения представляет собой просто область памяти, а объект — набор данных и методов работы с ними.

Метаданные

Метаданные используются для хранения информации о типах значений и ссылок, используемых в данном компоненте, в файл формата PE для дальнейшего использования этой информации в среде CLR. Эта информация используется для поиска и загрузки классов, размещения экземпляров этих классов в оперативной памяти, вызова их методов, трансляции IL-кода этих методов в машинный код, повышения безопасности работы с компонентом и установки границ контекста среды выполнения.

Использование метаданных является одним из основных отличий C# от C++. Компилятор C# создает метаданные из исходного кода и помещает их вместе с создаваемым им IL-кодом в PE-файл. Следует особо отметить, что эта операция производится стандартным образом, в отличие от компилятора C++, создающего собственные декорированные имена экспортируемых функций. Метаданные можно рассматривать как аналог библиотек типов в C++.

Метаданные каждого объекта среды .NET содержат следующую информацию:

- имя объекта;
- имена всех полей объекта и их типы;
- сигнатуры всех методов объекта.

Объединение исполняемого кода и метаданных позволяет исключить случаи отсутствия в распространяемом пакете информации о каком-либо из используемых в нем типов данных, а также разрешает проблемы с использованием различных версий библиотек, возникающие при использовании COM. Использование метаданных обеспечивает одновременную работу приложения с различными версиями одной и той же библиотеки, поскольку позволяет хранить соответствующую информацию за пределами системного реестра.

Обращение приложения к метаданным в процессе своей работы называется *рефлексией* (reflection). При этом приложение получает информацию о доступных объектах и об их членах. Рефлексия напоминает работу с интерфейсом COM `IDispatch`, но реализуется гораздо проще. Рефлексия не позволяет обеспечить безопасность типов, поэтому всю информацию из метаданных рекомендуется получать в процессе компиляции IL-кода. Одним из основных применений рефлексии является работа с шаблонами объектов, то есть с объектами, имена которых определяются в процессе выполнения программы.

Спецификация CLS

Спецификация Common Language Specification, сокращенно CLS, определяет набор типов VOS и содержит перечень соглашений, которым должна удовлетворять программа для своей совместимости с CLS. Эти соглашения гарантируют, что библиотека классов, соответствующая данной спецификации, может быть без проблем использована в любом приложении, написанном на любом языке, также соответствующем спецификации CLS. Следует обратить особое внимание на то, что требованиям спецификации CLS должны удовлетворять только интерфейсные элементы, то есть те элементы, доступ к которым возможен из другого компонента.

Такой подход позволяет вносить минимальные изменения в приложения, написанные на традиционных языках программирования, для обеспечения совместимости с данной спецификацией. При написании программы на C# не гарантируется ее автоматическое соответствие требованиям спецификации CLS, поэтому ниже будет приведен краткий обзор требований к типам и элементам языка, обеспечивающих удовлетворение этих требований.

- Простые типы.

  Можно использовать типы `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `object`, `short`, `string`.

- Массивы/

  Необходимо задать фиксированный размер массива (больший единицы). Он должен иметь нулевую нижнюю границу. Элементы массива должны принадлежать одному из перечисленных выше типов.

- Типы.

  Могут использоваться как абстрактные, так и завершенные типы. Могут реализовываться интерфейсы. Различные интерфейсы могут содержать методы, имеющие одинаковые сигнатуры. Не допускается множественное наследование типов. Элементы типов могут перегружаться и скрываться. Число и вид членов класса (полей, методов, конструкторов и т.д.) не ограничено. Класс вообще может не содержать ни одного члена. Частью интерфейса типа считаются только его открытые члены. Все типы значений, кроме перечислимых, должны быть потомками `System.ValueType`. Перечислимые типы должны быть потомками `System.Enum`.

- Члены типов.

  Член типа может перегружать или скрывать члены базовых типов. Типы аргументов и возвращаемых значений методов должны удовлетворять требованиям, предъявляемым CLS. Тип может иметь абстрактные члены.

- Методы.

  Методы могут быть статическими, виртуальными или методами объекта класса. Виртуальные методы и методы объекта класса могут иметь реализацию. В противном случае они рассматриваются как абстрактные. Статические методы всегда должны иметь реализацию.

- Поля.

  Поля могут быть статическими и динамическими. Статические поля могут содержать литеральную константу. В противном случае их значение должно задаваться в процессе инициализации.

- Свойства.

  Для доступа к свойствам должны в явном виде использоваться процедуры доступа `get` и `set` (использование синтаксиса работы со свойствами в данном случае недопустимо). Значение, возвращаемое процедурой доступа `get`, и первый аргумент процедуры доступа `set` должны иметь один и тот же тип CLS. Этот тип рассматривается как тип свойства. Все свойства должны иметь различные имена. Они не могут различаться только своим типом. Для доступа к свойствам следует использовать методы, имена которых имеют формат `get_PropertyName` и `set_PropertyName`, где `PropertyName` — имя свойства, к которому осуществляется доступ. Поскольку имена этих методов зарезервированы, тип, в котором определено это свойство, не может содержать методов с такими именами.

- Перечислимые типы.

  В качестве базовых типов для них могут использоваться только типы `byte`, `int`, `long` или `short`. Каждый элемент перечислимого типа должен представлять собой статическое поле, содержащее литерал базового типа. Перечислимые типы не могут иметь интерфейсы. Несколько полей перечислимого типа могут иметь одно и то же значение. Перечислимый тип должен быть потомком класса `System.Enum` (в C# это требование выполняется автоматически).

- Исключения.

  Допускается как вызов, так и обработка исключений. Пользовательские исключения должны быть потомками класса `System.Exception`.

- Интерфейсы.

  Допускается наследование интерфейсов. Интерфейс может содержать свойства, события и виртуальные методы. Реализация методов интерфейсов происходит в производных от них классах.

- События.

  Если в компоненте присутствует метод, добавляющий событие, тот же самый компонент должен содержать метод для его удаления. Каждый из этих методов должен содержать один формальный параметр — класс, производный от класса `System.Delegate`. Методы, используемые для работы с событиями, должны иметь имена, соответствующие следующему формату: `add_EventName`, `remove_EventName` и `raise_EventName`, где `EventName` — имя события.

- Пользовательские атрибуты.

  Могут использоваться только следующие типы: `bool`, `byte`, `char`, `double`, `enum` (базовым типом которого является один из типов CLR), `float`, `int`, `long`, `object`, `short`, `string`, `Type`.

- Объекты делегатов.

  Могут использоваться.

- Идентификаторы.

  Первый символ идентификатора должен принадлежать к определенному набору допустимых символов. Имена идентификаторов не могут различаться только регистром используемых в них символов (нечувствительны к регистру символов).

Приведенный список далеко не полон и содержит только минимально допустимый набор правил, позволяющих создавать простейшие приложения.

Виртуальная среда выполнения

Виртуальная среда выполнения реализована в системе *Virtual Execution System*. Эту среду можно рассматривать как реализацию Virtual Object System с помощью механизма *execution engine*, создающего среду выполнения NGWS.

Ниже перечислены основные компоненты Virtual Execution System:

- Язык Intermediate Language — достаточно подробно описанный выше промежуточный язык программирования. На этот язык транслируются приложения, написанные на языках программирования Visual Studio.NET.
- Загрузчик контролируемого кода — используется для разрешения конфликтов имен, размещения объектов в оперативной памяти и создания заглушек, позволяющих использовать JIT-компиляцию. При загрузке классов производится проверка их совместимости и контроль соблюдения в них определенных правил.
- JIT-компилятор — преобразует IL-код в машинный код.
- Загрузчик метаданных, проверяющий безопасность типов и целостность методов.
- Сборщик мусора и обработчик исключений — используются для работы со стеком. Контролируемый код позволяет производить трассировку стека в

процессе выполнения приложения. При этом стек разбивается на кадры, для работы с которыми компилятор исходного текста или JIT-компилятор должны создать специальный менеджер.

- Средства профилирования и отладки — используют информацию, предоставляемую компилятором исходного текста. Для использования этих средств компилятор исходного текста должен создать две таблицы, первая из которых используется для перевода конструкций исходного языка в адреса потока команд, а вторая — для перевода этого адреса в смещение в кадре стека. При работе JIT-компилятора содержимое этих таблиц меняется.
- Средства управления потоками и контекстами и средства для удаленной работы.

Зачем нужен C#

Из всего вышеизложенного можно сделать вывод о том, что создатели Visual Studio.NET поставили перед собой грандиозные задачи. Однако пока не очень понятно, зачем же создавать новый язык программирования. При описании некоторых свойств Visual Studio.NET упоминалось, что они наилучшим образом реализованы в языке C#, но тут же говорилось, что в этой среде все языки равноправны, поскольку приложения, написанные на любом из них, преобразуются в IL-код. Однако следует заметить, что любой из этих языков может использовать далеко не все функции промежуточного языка. Поэтому и появилась необходимость создания нового языка, в максимальной степени использующего возможности IL-кода.

При создании этого языка программирования преследовалось несколько целей:

- *Создать первый компонентно-ориентированный язык в семействе C/C++.* Как уже говорилось выше, программирование все в меньшей степени ориентируется на создание приложений, работающих на одном конкретном компьютере, и все больше смещается в сторону создания распределенных приложений, состоящих из компонентов, передаваемых по сети и подключаемых по мере необходимости. Для создания таких компонентов могут использоваться традиционные языки программирования (их и следует для этого использовать, поскольку они оптимизированы для решения подобных задач). Однако к языку, используемому для объединения этих компонентов в одно законченное приложение, предъявляются столь специфические требования, что им в полной мере не отвечает ни один из традиционных языков программирования. Этот язык не должен отличаться особой сложностью, поскольку все сложные задачи решаются на уровне компонентов, но должен обладать развитой системой диагностики и повышенным уровнем безопасности, так как ему приходится работать с объектами, совместная работа которых не гарантирована.

- *Создать альтернативу языку Jawa.* При создании языка Jawa, используемого для разработки приложений, которые должны работать везде "от кофеварки до сервера", выяснилось, что единственным путем создания такого языка является размещение всех используемых данных в объектах, то есть, грубо говоря, превращение всех типов данных в классы. Поскольку попытка "внести улучшения" в стандарт языка Jawa, похоже, бесславно завершилась, перед Microsoft встала задача создать собственную альтернативу данному языку, используя аналогичные подходы. Поэтому в C# не только сокращается разрыв между простыми типами и классами, но и имеется возможность создавать структурные типы, объектам которых не требуется выделять память в куче.
- *Повысить надежность программ и их устойчивость к ошибкам.* Для решения этой задачи в C# реализована автоматическая сборка мусора, исключающая возникновение утечек памяти, структурированная обработка исключений и безопасная работа с типами.
- *Создать максимально простой язык.* Как уже говорилось выше, на компонентно-ориентированном языке не предполагается создание особо сложных компонентов: для этого можно воспользоваться одним из уже имеющихся языков. Поэтому перед разработчиками C# была поставлена задача максимального упрощения языка при повышении его строгости. Для простоты подключения компонентов в C# интегрированы хорошо знакомые разработчикам механизмы COM и DLL, что также облегчает работу. В то же время, для создания и использования объектов C# не нужны файлы заголовков, файлы .idl или библиотеки типов. Эти компоненты содержат всю необходимую информацию в себе и не нуждаются в регистрации.

После того как мы ознакомились с целями, которые ставили перед собой разработчики языка программирования C#, посмотрим, каким образом им удалось реализовать эти цели, и что у них в результате получилось. Для этого рассмотрим основные свойства данного языка.

Простой

Язык C# одновременно является и очень простым, и очень сложным. Набор его исполняемых операторов сведен до минимума, а все используемые типы строго формализованы, что существенно облегчает его изучение. С другой стороны, этот язык предназначен для работы с достаточно сложными объектами и не приспособлен для выполнения простейших задач, обычно используемых для ознакомления с языком (кроме самых простых и, как правило, не имеющих практической ценности). Поэтому, чтобы эффективно использовать этот язык, необходимо разбираться в достаточно сложных областях программирования. Однако вернемся к его простоте.

Основным отличием C# от C и C++ является отсутствие указателей, что несколько непривычно после столь интенсивного их использования в этих языках. По умолчанию все программы на языке C# преобразуются в контролируемый код, в котором запрещено использование небезопасных операций, включая непосредственное обращение к памяти. Эта операция считается небезопасной, поскольку при ее выполнении отсутствует проверка того, что обращение производится именно к той области памяти, к которой хотел обратиться разработчик.

В C++ одни и те же операции `::`, `.` и `->` используются в пространствах имен, функциях-членах класса и ссылках. Для новичка бывает сложно разобраться, в каком контексте используется каждая из этих операций. В C# используется только одна операция `.` (точка), используемая для разделения имен при их вложении.

В C# используется *единая система типов*, рассматривающая любой тип, начиная от простейшего целочисленного типа до сложнейшего класса. Для того чтобы представление простого типа как объекта не привело к снижению быстродействия приложения, в C# используется механизм упаковки и распаковки.

С целью дальнейшего ужесточения проверки совместимости типов, целый и логический типы в C# разделены, что исключает проверку нулевого значения целочисленной переменной в условном операторе без формирования условного выражения, чем очень увлекаются программисты на C и C++. С другой стороны, это исключает появление ошибок при использовании операции присваивания вместо операции сравнения.

Удобный

При разработке C# был учтен обширный опыт работы с языком C++, и многие операции, которые в C++ приходилось выполнять самостоятельно, в C# выполняются автоматически.

Одним из нововведений C# является появление нового типа финансовых данных. Тип `decimal` позволяет программисту без всяких опасений сравнивать два числа, записанных в формате с плавающей точкой. Не секрет, что в C++ и других языках программирования эта операция, вследствие непредсказуемых ошибок округления, была довольно-таки рискованной. Если и этот тип вас не устраивает, вы можете создать свой собственный простой тип, удовлетворяющий всем вашим запросам (а если не сможете, значит, ваши запросы превышают ваши возможности как программиста).

Как уже неоднократно говорилось выше, в C# реализовано автоматическое управление памятью: выделение памяти осуществляется пользователем, а ее уничтожение — сборщиком мусора. Поскольку ссылка на производный класс может быть оформлена как ссылка на его базовый класс, и у про-

граммиста нет возможности в явном виде указать тип уничтожаемого объекта, подобное распределение обязанностей подразумевает существенное повышение требований к безопасности используемых типов.

Надежный и безопасный

Одной из основных задач при разработке среды Visual Studio.NET было повышение надежности и безопасности создаваемых в ней приложений. Одним из средств решения этой задачи является отказ от использования указателей и исключение пользователя из процесса управления памятью. Опасность использования указателей связана с тем, что при применении явного преобразования типов можно произвести преобразование любого указателя на любой тип в указатель на другой тип. Например, компилятор не выдал ошибки при явном преобразовании указателя на потомка класса `CWinApp` к типу `double*`. Конечно, за все явные преобразования типов отвечает разработчик, но стоит ли возлагать на него лишнюю ответственность.

В C# программист также отвечал за освобождение всей выделенной ему динамической памяти. Казалось бы, это позволяло ему оптимальным образом работать с памятью, уничтожая объект сразу же, как только в нем отпадала необходимость. Однако этой возможностью пользовались немногие: локальные переменные нет никакого смысла делать динамическими, а память под динамические переменные класса, как правило, выделялась в его конструкторе и освобождалась в его деструкторе. Подобный подход к освобождению памяти, выделенной под динамические переменные класса, может быть легко автоматизирован, что и сделано в C#. Автоматическое освобождение памяти избавляет разработчика от необходимости отслеживать уничтожение всех динамических переменных класса, что не только освобождает его от части хлопот, но и существенно снижает вероятность возникновения утечек памяти.

Ярким примером небезопасности Visual C++ является преобразование типа целого числа со знаком в целое число без знака и обратно. Известно, что Microsoft испытывает патологическую любовь к целым числам без знака и повсеместно использует их в контексте, где уместнее было бы использовать целое число со знаком. Например, в MFC определен класс массива изменяемого размера для целых чисел без знака и не определен подобный массив для целых чисел со знаком. Как выяснилось, эта проблема решается элементарно просто: при присваивании переменной, имеющей тип целого со знаком, значения, имеющего тип целого без знака, производится побитовое копирование. То есть значения, превышающие максимальное значение целого числа со знаком, рассматриваются как отрицательные. То же самое происходит и при обратном присваивании: отрицательное значение преобразуется в соответствующее положительное значение. Такое преобразование удобно, но противоречит всем правилам преобразования типов.

Среда выполнения программ C# является верифицируемой, то есть в процессе выполнения приложения она следит за тем, чтобы работающий код не производил недопустимых операций с типами. При этом производится не только проверка допустимости преобразований типов (например, при передаче аргумента методу), но и проверка допустимости обращения по указанному адресу.

Требования безопасности накладывают на разработчика определенные ограничения, основные из которых перечислены ниже.

- Все переменные должны быть инициализированы. Все объекты в C# инициализируются по умолчанию. В результате этой инициализации вся память, выделенная под данные инициализируемого объекта, заполняется нулевыми значениями. Локальные переменные должны инициализироваться разработчиком. При появлении в программе неинициализированной переменной компилятор выдаст сообщение об ошибке.
- В C# введены строгие правила преобразования типов: тип объекта может быть преобразован к другому типу только в том случае, если тип этого объекта является производным от типа, к которому производится преобразование, или же в том случае, если такое преобразование восстанавливает тип объекта, измененный при преобразовании к базовому типу. Следует помнить, что эти правила действуют при непосредственном преобразовании типа. При присваивании типа одного объекта другому действуют более сложные правила, сводящиеся к нахождению для этих объектов общего типа, к которому может быть произведено преобразование, и присваиванию объекту этого типа.
- В C# производится проверка границ выделенной памяти. Поэтому в массив может быть записано только то число элементов, которое указано при его создании. При попытке записи "дополнительного" элемента будет выдано сообщение об ошибке.
- В C# определены строгие правила выполнения арифметических операций. При возникновении переполнения в некоторых случаях генерируется исключение (как правило, при работе с целыми числами), а в некоторых — результатом операции является специальное значение, которое может быть использовано в последующих операциях (как правило, при работе с числами с плавающей точкой).
- В C# осуществляется контроль типов аргументов, передаваемых по ссылке.

Совместимый

Язык C# изначально создавался для объединения компонентов, разработанных с использованием различных языков программирования. Visual Studio.NET позволяет использовать в программах C# вызовы функций из библиотек динамической компоновки или использовать компоненты COM, которые рассмат-

риваются ею как компоненты .NET Runtime. Эти компоненты взаимозаменяемы — не только компоненты COM могут представляться как компоненты .NET Runtime, но и компоненты NET Runtime могут использоваться в качестве компонентов COM.

Простота языка программирования C# имеет одно негативное следствие: на нем сложно написать серьезное приложение. Это обстоятельство привело к тому, что в среде программирования, используемой этим языком, предусмотрены мощные средства межъязыковой интеграции. Так, например, класс, написанный на Visual Basic (или любом другом языке программирования Visual Studio.NET), может использоваться в качестве базового для создания класса языка C#, который, в свою очередь, может, например, использоваться классом C++. Таким образом, ответить на вопрос, на каком языке написан класс, становится в некоторых случаях очень затруднительным.

Решение задачи тесной интеграции языков наталкивается на определенные сложности, связанные с тем, что некоторые типы данных одного языка не имеют непосредственных аналогов в другом языке. Эта задача решается введением спецификации CLS, соответствие которой обеспечивает беспрепятственное использование компонента языками среды Visual Studio.NET. Эта спецификация содержит требования к открытому интерфейсу класса, в который, например, запрещено включать данные, имеющие тип целых чисел без знака, поскольку они не поддерживаются некоторыми языками.

При компиляции приложения на C# можно указать, что создаваемый код должен соответствовать спецификации CLS. В этом случае компилятор пометит все несовместимые участки.

Среда выполнения .NET Runtime использует единый механизм обработки ошибок, основанный на исключениях. Это вызвано необходимостью унификации разнообразных механизмов обработки ошибок, используемых в имеющихся библиотеках. Теперь все коды ошибок и значения `HRESULT` преобразуются в исключения и обрабатываются соответствующим образом.

Все основные функции, традиционно находящиеся в библиотеках времени исполнения (runtime), а также множество новых, объединены в библиотеки BCL (Base Class Libraries). В эти библиотеки, помимо прочего, входят:

- классы коллекций, реализующие очереди, массивы изменяемого размера, стеки и другие программные элементы;
- классы для работы с базами данных;
- классы ввода/вывода;
- классы WinForm, облегчающие создание пользовательского интерфейса;
- классы для работы в сетях.

Объектно-ориентированный

Хотя C# является компонентно-ориентированным языком программирования, он в полной степени является и объектно-ориентированным языком. Это вполне естественно, поскольку развитие в данном случае не предполагает отрицания (согласно положениям марксистско-ленинской философии это свидетельствует о том, что переход от объектно-ориентированных языков к компонентно-ориентированным не является революционным).

Строгое следование принципам объектно-ориентированного программирования заставило отказаться в C# от глобальных функций, переменных или констант. Все эти элементы могут быть только членами какого-либо класса (статическими или нестатическими). Подобный подход улучшает читабельность программ C# и снижает вероятность возникновения конфликтов имен.

Для устранения возможности случайной перегрузки методов класса, они по умолчанию не являются виртуальными. При объявлении в производном классе метода с той же сигнатурой метод базового класса скрывается, и вместо него используется метод производного класса. Чтобы эта операция не прошла незамеченной, компилятор C# выдает предупреждение, для подавления которого следует использовать модификатор `new`. Для объявления виртуального метода необходимо в явном виде указать для него модификатор `virtual`, а при его перегрузке — указать модификатор `override`. Таким образом, весь процесс наследования методов находится под постоянным контролем среды программирования.

В C# расширен набор модификаторов права доступа. Кроме уже знакомых по C++ модификаторов `private`, `protected` и `public`, в нем используется модификатор `internal`. Другой особенностью C# является отказ от множественного наследования для классов. Эта возможность оставлена только для интерфейсов.

Совместимый с унаследованными технологиями

Все описанное выше выглядит замечательно, но сразу же возникает вопрос, что же делать с теми компонентами, которые не соответствуют требованиям, предъявляемым средой выполнения. К ним, прежде всего, относятся различные API, требующие передачи указателя. Их слишком много, чтобы от них просто отмахнуться. Та же проблема возникает и при обращении к "родным" функциям Win32. В этом случае приходится использовать небезопасный код (в некоторых случаях эта проблема может быть решена с использованием COM и PInvoke).

Для взаимодействия с такими API и функциями в C# создаются небезопасные классы и методы, в которых допускается использование указателей,

структур и статических массивов. Небезопасный код, так же как и безопасный, выполняется в контролируемом пространстве. Небезопасные переменные помещаются в блок памяти, контролируемый сборщиком мусора. Поэтому единственным их отличием от безопасных переменных является невозможность их перемещения сборщиком мусора.

Контролирующий версии

Каждый разработчик, создав удачный программный продукт, заботится о его дальнейшей поддержке, которая выражается в исправлении выявленных ошибок и расширении набора его функциональных возможностей. Некоторые разработчики (например, Microsoft) положили этот процесс в основу своей коммерческой стратегии, сделав основной упор не на исправлении ошибок, а на расширении набора функциональных возможностей. При этом максимальная несовместимость старых версий с новыми заставляет пользователя часто обновлять версии используемых им программных продуктов.

Исправление ошибок, конечно, является благим делом и заслуживает всяческого поощрения. Иначе обстоит дело с расширением набора функциональных возможностей. Дело в том, что эти возможности, как правило, не берутся с потолка, а являются отражением потребностей пользователя, которые возникают задолго до того, как их решили включить в новую версию программного продукта. Поэтому существует большая вероятность, что разработчик, использующий этот программный продукт, уже реализовал у себя эти функциональные возможности. Поскольку разработчики стремятся давать своим классам и методам мнемонические имена, связанные с областью их применения, а набор этих имен существенно ограничен, то возникновение конфликта имен, используемых разработчиком и имен, введенных в новую версию приложения, весьма вероятно.

Для разрешения этих конфликтов (скорее для информирования разработчика об их возникновении) в CLR встроен контроль версий используемых компонентов. Благодаря такому контролю разработчик может быть уверен, что его приложение сохранит совместимость с новыми версиями используемых в нем компонентов, и новые версии созданного им компонента могут быть без проблем применены пользователями в своих приложениях.

Легко распространяемый

Поскольку C# разрабатывался как язык для создания распределенных приложений, в него не могла быть не включена поддержка распространения компонентов. С использованием CLR процесс распространения приложений максимально упростился. В некоторых случаях он даже не включает в себя этап установки распространяемого приложения. Использование метаданных позволяет использовать распространяемый компонент в любой сре-

де с поддержкой .NET и обеспечить приложению работу с теми компонентами, с которыми оно распространялось, а не с другими версиями тех же компонентов, которые могут быть установлены на том же компьютере.

До появления среды CLR распространяемый программный пакет мог включать в себя исполняемый файл приложения, файлы библиотек динамической компоновки и файлы их описания (.lib), файлы объектных библиотек COM, библиотеки типов и другие файлы.

В среде CLR компоненты распространяются в виде *сборки* (assembly). Как уже говорилось выше, результатом компиляции программы на языке C# является IL-код. Сборка содержит этот IL-код и связанные с ним метаданные и различные файлы, необходимые для работы распространяемого компонента на удаленном компьютере. Каждая сборка содержит *манифест* (manifest), включающий в себя перечень файлов, входящих в нее, перечень экспортируемых типов и ресурсов, а также таблицу соответствия этих типов и ресурсов файлам, содержащим их реализацию. Помимо этого, в манифесте указаны другие сборки, наличие которых необходимо для корректного функционирования данной сборки. За исключением взаимозависимости сборок каждая сборка является полностью автономной. Если сборка состоит из нескольких файлов, возможна ситуация, когда отдельные файлы сборки будут подгружаться по мере необходимости.

Глава 2

Создание простейшего приложения

Опытный программист при встрече с новым языком программирования старается сначала понять, на какой из известных ему языков он больше всего похож, и что из других языков в нем используется. Поскольку все языки программирования имеют между собой много общего, он может разобраться в простейших программах, написанных практически на любом языке, не зная его основ. С другой стороны, основы языка лучше всего продемонстрировать на конкретных примерах.

Хотя данная книга рассматривает язык C#, не привязываясь к его конкретной реализации, для создания приложения пользователю нужно работать в какой-то среде программирования. В настоящее время единственной средой программирования, позволяющей работать с языком C#, является пакет Microsoft Visual Studio.NET. Более полное описание этой среды программирования приведено в приложении к данной книге.

Чтобы создать консольное приложение в пакете Microsoft Visual Studio.NET:

1. Выберите команду **File**, **New**, **Project...** (Файл, Создать, Проект). Появится диалоговое окно **New Project** (Новый проект), изображенное на рис. 2.1.

Рис. 2.1. Диалоговое окно **New Project**

2. В окне иерархической структуры **Project Types:** (Типы проектов). Раскройте папку **C# Projects** (Проекты C#). Диалоговое окно **New Project** примет вид, изображенный на рис. 2.2.

Рис. 2.2. Диалоговое окно **New Project**, папка **C# Projects**

3. В текстовое поле **Name:** введите новое имя проекта **Hello**.
4. В окне списка дважды щелкните левой кнопкой мыши на значке **Console Application** (Консольное приложение). Диалоговое окно **New Project** закроется и будет создана заготовка консольного приложения.
5. В открытом окне редактирования файла Class1.cs замените текст заготовки консольного приложения следующим текстом.

```
namespace Hello
{
using System;

class Hello
{
    static void Main()
    {
            Console.WriteLine("Hello, I'm C#");
    }
}
}
```

6. Выберите команду **Debug**, **Start** или нажмите клавишу **<F5>**. Приложение будет откомпилировано и запущено на исполнение. В результате этого на экране на короткое время появится окно с текстом **Hello, I'm C#**.

По умолчанию файлы с текстами на языке C# имеют расширение cs. Кроме приведенного выше файла, проект `Hello` содержит еще один файл с этим расширением, в котором устанавливаются значения атрибутов. В него не нужно вносить изменений, поэтому пока он не будет рассматриваться.

В CLS, а, следовательно, и в C#, определяется единое пространство имен для всей информации (и метаданных) программы. В данном случае это пространство имен `Hello`. Пространства имен используются для иерархической организации программ, написанных на языке C#, и для компоновки элементов библиотек классов.

Для упрощения доступа к системным классам в этом пространстве имен используется директива `using System`, которая позволяет использовать простые имена для элементов пространства имен `System`, определенного в библиотеке классов Microsoft.NET. Это пространство имен содержит, помимо прочего, описание используемого в данной программе класса `Console`.

Метод Main

Точкой входа в любое приложение на зыке C# является метод `Main`, принадлежащий одному из классов или структур этого приложения (в данном случае — классу `Hello`). В этом методе начинается и завершается работа приложения. Метод `Main` должен быть открытым статическим методом (то есть иметь модификаторы `public` и `static`). Имя этого метода пишется с заглавной буквы, а не со строчной, как в C++.

Объявление статического метода в качестве входной точки приложения обеспечивает ее уникальность, так как статический метод принадлежит всему классу, а не каждому его объекту. Поскольку метод `Main` является обычным методом класса, он может быть вызван в любом месте программы как `Hello.Main`, однако это приведет к рекурсии, чреватой переполнением стека. Потому использование такого вызова настоятельно не рекомендуется.

Помещение главной функции приложения в класс связано с тем, что в C# отсутствуют глобальные функции и переменные. Они могут быть только членами классов или структур. Входная точка приложения должна быть уникальной, поэтому, казалось бы, метод с именем `Main` может содержаться только в одном классе или структуре приложения, но это не так.

Одним из способов тестирования объектов является включение в их описание специальной функции, проверяющей правильность их работы. В C# такую функцию можно оформить в виде метода `Main`, что существенно облегчает ее запуск на исполнение. Однако в этом случае компилятору необходимо в явном виде указать, какой из методов `Main` является входной точкой приложения. Для этого используется ключ компилятора `/main:`, имеющий синтаксис `/main:<имя класса>` (Например, `/main:<Hello>`).

В приведенном примере метод `Main` не возвращает значения (то есть возвращает значение типа `void`), однако, возможен вариант, когда он будет возвращать целочисленное значение, в котором, как правило, передается код ошибки (так что заявление о том, что в CLR все ошибки обрабатываются только как исключения, несколько опрометчиво). Кроме того, метод `Main` может иметь формальный параметр, имеющий тип массива строк. В этом формальном параметре ему передаются аргументы командной строки приложения.

Как правило, для проверки того, что данной программе были переданы аргументы командной строки, проверяется значение свойства `Length` строкового массива. Например, если указание аргумента в методе `Main`, возвращающем значение, обязательно, проверка может производиться следующим образом:

```
public static int Main( string[] args )
{
        if (args.Length == 0)
        {
                Console.WriteLine("Please enter an argument.");
                return 1;
        }
        ...
        return 0;
}
```

Самое интересное в этом примере то, что даже в том случае, когда метод `Main` возвращает значение 1, в окне **Output** сообщается, что соответствующее приложение завершило свою работу с кодом 0.

Для преобразования строковых аргументов в числовую форму используется метод `Parse`. Например, для получения значения единственного целочисленного формального параметра может быть использовано следующее выражение: `int num = int.Parse(args[0])`.

Работа с консолью

В главном методе приложения `Main` вызывается статический метод `System.Console.WriteLine`. Поскольку вызванная до этого директива `using` предоставила доступ к пространству имен `System`, имя данного метода было сокращено до `Console.WriteLine`.

Для вывода информации на экран данное приложение использует библиотеку классов, однако, как уже говорилось выше, C# не содержит собственных библиотек классов, поэтому он вынужден использовать библиотеки классов других языков таких, как Visual Basic и Visual C++. Единственным исключением являются функции для работы с консолью.

Поскольку на языке C# предполагается писать простейшие приложения, объединяющие в одно целое сложные компоненты, написанные на других языках программирования, нет необходимости введения сложных функций для вывода информации на экран. От него требуется только вывести на экран некоторое сообщение о своем состоянии и получить краткое руководящее указание по своим последующим действиям. Для этого достаточно введения в этот язык функций для работы с системной консолью.

В языке C для работы с системной консолью использовались функции `scanf` и `printf`, а также различные их модификации, в C++ помимо этих функций могли быть использованы потоки `cin` и `cout`, существенно упрощавшие этот процесс. В C# для этого используется упомянутый выше класс `Console`.

Помимо описанного метода `WriteLine` класс `Console` содержит метод `Write`, отличающийся от него тем, что при выводе строки данная функция автоматически не добавляет в ее конец символы возврата каретки и перевода строки, что позволяет выводить строку по частям. Для ввода информации с консоли используется функция `ReadLine`. Для демонстрации работы этих функций внесем изменения в рассмотренное выше приложение.

Для демонстрации работы консольных функций:

1. Измените содержимое файла Class1.cs в соответствии с приведенным ниже текстом:

```
namespace Hello
{
using System;

class Hello
{
    static void Main()
    {
            Console.Write("My name is ");
            string MyName = Console.ReadLine();
            Console.WriteLine("Hello {0}", MyName);
    }
}
}
```

2. Нажмите клавишу **<F5>** и запустите приложение на исполнение. Появится окно консольного приложения, предлагающее вам представиться.
3. Введите свое имя и нажмите клавишу **<Enter>**. Приложение вас поприветствует, как это показано на рис. 2.3.

Рис. 2.3. Окно консольного приложения

Единственное, что нуждается в пояснении, это формат аргументов функции `WriteLine`. В ней использован вывод форматированной строки. Первый аргумент функции содержит выводимую строку, в которой с помощью специальных символов форматирования указано место вывода следующих аргументов. Специальный символ представляет собой пару фигурных скобок, в которые помещается номер аргумента, предназначенного для размещения в данной позиции. Все аргументы функции `WriteLine` нумеруются с нуля, не считая выводимой строки с символами форматирования. При необходимости выводимые аргументы автоматически преобразуются в текстовый вид.

Английский язык при выводе информации использовался потому, что в применявшейся автором версии языка C# имелись проблемы с выводом кириллицы.

Включение комментариев

При создании приложения хорошим тоном считается его полное документирование, с включением, помимо прочего, комментариев в исходный текст программы. Поскольку комментарии не компилируются, при их написании можно использовать кириллицу.

Формат комментариев C# полностью повторяет формат комментариев C++. Строковые комментарии начинаются с комбинации символов `//` и, как следует из их названия, завершаются с концом строки, в которой они размещены. Эти комментарии могут занимать всю строку или располагаться в конце строк, содержащих другую информацию.

Если комментарий не может быть размещен на одной строке, то его можно оформить как последовательность строковых комментариев (что очень часто делается исходя из эстетических соображений), или же поместить в специальные скобки. В последнем случае открывающая скобка комментария представляет собой комбинацию символов `/*`, а закрывающая скобка ком-

ментария — комбинацию символов `*/`. Такие комментарии называются *регулярными*. Следует помнить о том, что скобки регулярных комментариев не допускают вложения, то есть текст, расположенный в этих скобках не может содержать комбинации символов `*/`, поскольку это приведет к появлению неоднозначности.

Чтобы продемонстрировать использование различных типов комментариев, включим их в рассмотренное ранее приложение. В результате содержимое файла Class1.cs примет следующий вид:

```
namespace Hello
{
using System;  // Получение доступа к пространству имен System

// Главный класс приложения

class Hello
{
        /* Входная точка приложения
          может возвращать код ошибки
          и обрабатывать аргументы командной строки */

        static void Main()
        {
                Console.Write("My name is ");              // Вывод
                                                           // подсказки
                string MyName = Console.ReadLine(); // Ввод имени
                Console.WriteLine("Hello {0}", MyName);    // Вывод
                                                           // приветствия
        }
}
}
```

При выводе регулярного комментария среда программирования Visual Studio.NET автоматически ставит в начале каждой его строки символ звездочки (`*`). Поскольку это не требуется синтаксисом комментария, все эти символы были удалены.

Глава 3

Первое знакомство с C#

Язык программирования C# (произносится как "Си шарп") представляет собой новый объектно-ориентированный язык, обеспечивающий высокий уровень проверки типов переменных в исходных текстах программ. Хотя Microsoft желал бы представить этот язык, как продолжение линии C и C++, сочетающий в себе высокую производительность Visual Basic и мощь C++, язык C# является, по сути, попыткой создать патенточистую альтернативу языку Java, все права на который принадлежат корпорации Sun. В этой главе мы кратко ознакомимся с основными особенностями C#.

Автоматическое управление памятью

При работе с языками программирования C и C++ программисту приходилось самому заботиться о выделении памяти для динамических переменных и об ее последующем освобождении. Это было не только обременительно, но и служило источником для утечек памяти, потенциально приводящих к серьезным системным ошибкам.

Конечно, для какого-либо критичного приложения разработчик может создать схему управления распределением памяти, намного превосходящую по своим возможностям стандартные сборщики мусора. Но это справедливо только для указанной области, то есть для небольших приложений, выполняющих операции, в которых разработанная схема управления памятью может дать значительный эффект. В большинстве же случаев, особенно при разработке сложных приложений, использование нестандартных схем работы с памятью не приводит к существенному повышению их эффективности, но служит причиной возникновения множества проблем. Для борьбы с этими проблемами разработчику при работе с памятью приходится придерживаться стратегии, практически не отличающейся от стратегии, используемой сборщиками мусора. Поэтому в C# используется *автоматическое управление памятью.*

Выделение памяти из кучи в среде CLR производится предельно просто и эффективно. Среда убеждается, что в управляемой куче имеется фрагмент свободной памяти, в котором мог бы разместиться создаваемый объект, возвращает указатель на начало этой области памяти и переводит указатель свободного фрагмента на адрес первой свободной ячейки памяти после созданного объекта.

Если в управляемой куче отсутствует фрагмент свободной памяти требуемого размера, включается механизм сборки мусора, удаляющий объекты, на которые в данный момент отсутствуют ссылки. Для повышения эффективности данной процедуры объекты, имеющие размер более 20 КБ, помещаются в отдельную кучу.

Работа сборщика мусора CLR основана на алгоритме "пометки и сжатия". В соответствии с этим алгоритмом сборщик мусора просматривает всю иерархию размещенных в куче объектов, начиная с терминальных, и помечает те из них, на которые имеются ссылки. Все остальные объекты удаляются.

После удаления объектов производится дефрагментация кучи. При этом все оставшиеся объекты перемещаются в начало кучи, и производится соответствующая модификация указателей на них. Поскольку при перемещении объектов ссылки на них становятся некорректными, на время работы сборщика мусора вся другая работа в системе прекращается.

Одновременный просмотр всех объектов (включая глобальные, статические и локальные объекты, а также регистры процессора) приводит к большому перебору, многие операции которого, скорее всего, окажутся излишними, поскольку можно предсказать, что если объект пережил уже несколько операций сборки мусора, он, скорее всего, переживет и данную. Это связано с тем, что чем выше уровень объекта в иерархии, тем дольше срок его жизни. Понятно, что объект класса существует дольше, чем локальные объекты, определенные в его методах.

Поэтому все объекты, просматриваемые сборщиком мусора CLR, делятся на три поколения. К поколению 0 относятся объекты, не прошедшие еще ни одной сборки мусора. К поколению 1 относятся объекты, прошедшие одну сборку мусора, а к поколению 2 — объекты, прошедшие несколько сборок мусора.

Операция сборки мусора начинается с просмотра объектов поколения 0. Если просмотра этих объектов достаточно, чтобы освободить требуемый объем памяти, операция сборки мусора на этом заканчивается. Если объем выделенной памяти недостаточен, начинается просмотр объектов поколения 1. И только в том случае, если просмотр этих объектов не привел к освобождению требуемого объема памяти, просматриваются объекты поколения 2. Объект считается прошедшим сборку мусора только в том случае, если он был в ней просмотрен.

Поскольку при каждой сборке мусора производится дефрагментация кучи, в ней сначала располагаются все объекты поколения 2, так как они были созданы ранее всех объектов других поколений, затем объекты поколения 1 и только потом — объекты поколения 0, поскольку они создавались после последней дефрагментации кучи.

Автоматическое управление памятью не означает того, что пользователь не может создать деструктора объекта и поместить в него некоторые операции,

которые необходимо произвести при его уничтожении. Однако следует помнить, что деструкторы C# существенно отличаются от деструкторов C++. При создании объекта C#, имеющих деструктор среда CLR включает ссылку на него в список объектов, нуждающихся в завершении. Если на объект, включенный в этот список, отсутствуют ссылки, считается, что он готов к завершению.

После окончания процедуры сборки мусора активизируется поток завершения, в котором вызываются деструкторы для всех объектов, готовых к завершению. После работы деструктора ссылка на этот объект удаляется из списка объектов, нуждающихся в завершении, и при следующей сборке мусора этот объект может быть удален.

Процессом сборки мусора можно управлять, хотя это и может вызвать проблемы в других фрагментах программы. Для немедленной инициализации процедуры сборки мусора вызывается метод `System.Collect`. Вызов этого метода оправдан в том случае, если разработчик уверен, что перед этим было освобождено большое число объектов.

В качестве примера автоматического управления памятью рассмотрим фрагмент приложения, сохраняющего информацию в стеке. Текст этого фрагмента приведен ниже:

```
using System;
public class Stack
{
        privateElementCurrent= null;

        public bool Is_Empty
        {
                get
                {
                        return (Current == null);
                }
        }

        public object Pop_Element()
        {
                if (Current == null)
                        throw new Exception("Stack is empty.");
                else
                {
                        object val     = Current.Value;
                        Current        = Current.Link;
```

```
                return val;
            }
        }

        public void Push_Element(object val)
        {
            Current = new Element(val, Current);
        }

        class Element
        {
            // Переменные класса
            public ElementLink;
            public object Value;
            // Конструкторы класса
            public Element(object new_value): this(new_value, null)
        {}
            public Element(object new_value, Element next_value )
            {
                Link   = next_value;
                Value  = new_value;
            }
        }
}
```

В рассмотренном примере класс `Stack` реализует связный список объектов класса `Element`. Корневой элемент стека располагается в переменной `Current` класса `Stack`. Извлечение элемента из стека производится в функции `Pop_Element` этого класса. Если стек пуст, функция вызывает исключение, в противном случае данная функция возвращает содержимое корневого элемента стека и корневым элементом становится предыдущий элемент стека. Новые объекты класса `Element` создаются в функции `Push_Element`.

Обратите внимание на то, что объекты класса `Element` в приведенном фрагменте только создаются, но нигде не уничтожаются. Уничтожение объекта класса происходит в том случае, когда программа уже никоим образом не может получить к нему доступ.

Для проверки работы приведенного выше фрагмента программы его следует включить в текст консольного приложения, создание которого описано в главе 2, и изменить главную функцию приложения в соответствии со следующим текстом:

```
        static void Main()
        {
```

```
        Stack st = new Stack();

        for (int n = 0; n < 12; n++)
                st.Push_Element(n);

        while (!st.Is_Empty)
                Console.WriteLine(st.Pop_Element());
    }
```

См. главу 2.

В данном примере в стек записываются 12 элементов. После этого стек очищается. Все 12 элементов могут уничтожаться или перемещаться сборщиками мусора только после закрытия объекта класса `Stack`, однако, никто не гарантирует, что они будут сразу же после этого уничтожены. Такое вольное обращение с памятью может привести к ее существенному перерасходу.

В тех случаях, когда использование безопасных методов работы с типами приводит к существенному снижению производительности приложения, C# позволяет включать в программы так называемые небезопасные (unsafe) фрагменты. В них разрешается непосредственно использовать типы указателей на объекты. Присваивание переменой этого типа указателя на объект запрещает сборщикам мусора перемещать объект, что повышает быстродействие приложения. Поскольку в небезопасных фрагментах выполняются операции, недопустимые в других фрагментах программы, они должны маркироваться модификатором `unsafe`. Рассмотрим это на примере:

```
using System;

class Swift
{
    unsafe static void PrintArray(int[] Array)
    {
        fixed (int *lpArray = Array)
        {
            int    *lpElement = lpArray;

            for (int i = 0; i < Array.Length; i++)
            {
                int val = *lpElement;
                Console.WriteLine( "Element {0} value {1}",
                i, val );
                lpElement++;
            }
```

```
            }
        }

        static void Main()
        {
            int[] Array = new int[16];

            for(int i=0; i < 16; i++ )
                Array[i] = i;

            PrintArray( Array );
        }
}
```

Типы переменных

В C# поддерживается стандартный набор типов данных. Однако как уже говорилось выше, каждый тип данных в C# является объектом, что не предусматривалось для большинства типов данных, входящих в стандартный набор. Для разрешения этого противоречия в C# различают два основных типа переменных: *типы значений и типы ссылок*. Типы значений включают в себя простейшие типы переменных, а также типы перечисляемых данных и структур, то есть те типы, которые обычно не являются объектами. К простейшим типам переменных относятся, например, такие типы, как `char`, `int` или `float`. Чтобы превратить их в объекты и, одновременно обеспечить их использование в тех языках, где они не являются объектами, каждому простейшему типу в C# сопоставляется объект CLS.

К типам ссылок относятся типы классов, интерфейсов, массивов и делегатов. Основное отличие типов значений от типов ссылок заключается в том, что переменные типов значений используются для хранения одного или нескольких значений некоторых величин, а переменные типов ссылок хранят ссылки на объекты.

Предопределенные типы переменных

Большинство предопределенных типов переменных, используемых в C#, используются в C и C++. Как правило, это простейшие типы переменных. Для ссылок предопределены всего два типа: объект и строка.

Описание предопределенных типов сведено в табл. 3.1.

Таблица 3.1. *Предопределенные типы переменных*

Тип	Описание	Пример
`object`	Базовый класс для всех остальных типов	`object obj = new Stack();`
`string`	Строковый тип, последовательность символов Unicode	`string sz = "Привет";`
`sbyte`	8-разрядное целое число со знаком	`sbyte sbVal = 5;`
`short`	16-разрядное целое число со знаком	`short sVal = 5;`
`int`	32-разрядное целое число со знаком	`int iVal = 5;`
`long`	64-разрядное целое число со знаком	`long lVval = 5;` `long Val = 5L;`
`byte`	8-разрядное целое число без знака	`byte bVal = 5;` `byte Val = 5U;`
`ushort`	16-разрядное целое число без знака	`ushort usVal = 5;` `ushort Val = 5U;`
`uint`	32-разрядное целое число без знака	`uint uVal = 5;` `uint Val = 5U;`

`ulong`	64-разрядное целое число без знака	`ulong ulVal = 5;`
		`ulong Val = 5U;`
		`ulong V = 5L;`
		`ulong Vl = 5UL;`

`float`	Число с плавающей запятой единичной точности	`float Val = 3.1416926F;`
`double`	Число с плавающей запятой двойной точтости	`double dVal = 1.23;` `double Val = 8.9D;`
`bool`	Логический тип переменной. Может принимать значения true или false.	`Bool bVal = true;`
`char`	Тип символа Unicode	`char cVal = 'z';`
`decimal`	Тип десятичного числа с 28 значащими цифрами	`decimal dVal = 3.1415926M;`

Каждый предопределенный тип является сокращением некоторого системного типа. Например, тип `int` представляет собой альтернативное имя структуры `System.Int32`. В большинстве случаев оба эти имени полностью равноправны, но удобнее и привычнее использовать сокращенные имена.

Массивы

В C# можно создавать как одномерные, так и многомерные массивы, причем в качестве многомерных массивов могут выступать как обычные массивы, для которых задаются размеры для каждого измерения, так и иерархические массивы, представляющие собой массивы указателей на массивы.

Чаще всего используются одномерные массивы. Тип массива получается добавлением к типу элементов массива одной или нескольких пар квадратных скобок. Скобки располагаются справа от типа элемента массива. Ниже приведено несколько примеров объявления массивов:

```
class Demo
{
        static void Main()
        {
                byte[]          b1;     // одномерный байтовый массив
                byte[,]         b2;     // двумерный байтовый массив
                byte[,,]        b3;     // трехмерный байтовый массив
                byte[][]        pb2;    // массив указателей на одномерные
                                        // байтовые массивы
                byte[][,]       pb3;    // массив указателей на двумерные
                                        // байтовые массивы
        }
}
```

Тип массива относится к типам ссылок, поэтому после объявления массива его следует еще инициализировать. Инициализация массива может быть включена в его объявление. При этом можно не только задать размерность массива, но и инициализировать его элементы. Ниже приведено несколько примеров инициализации массивов:

```
class Demo
{
        static void Main()
        {
                byte[]          b1      = new byte[] {1, 2, 3};
                byte[,]         b2      = new byte[,] {4, 5, 6}, {7, 8, 9}};
                byte[,,]        b3      = new byte[3, 3, 3];
                byte[][]        pb2     = new byte[2][];
                byte[][,]       pb3     = new byte[3][,];

                pb2[0] = new byte[] {1, 2};
```

```
            pb2[1] = new byte[] {3, 4, 5, 6, 7};
            pb3[0] = new byte[,] {{5, 6},{7,8}}};
            pb[1]  = new byte[3, 4];
            pb[2]  = new byte[20, 30];
        }
}
```

В этом примере приведены различные варианты инициализации массивов и то, что элементы данного массива были инициированы при его создании, совсем не означает , что они не могут быть изменены в процессе выполнения программы. При инициализации элементов обычных массивов необходимо помнить, что при этом следует использовать блоки данных одинакового размера и структуры, поскольку только иерархические массивы могут состоять из указателей на массивы различного размера (но не размерности).

При определении локальных переменных разрешается использовать упрощенную форму инициализации элементов массива и не указывать повторно тип массива, то есть вместо выражения

```
        byte[] b1     = new byte[] {1, 2, 3};
```

в предыдущем примере можно было бы использовать выражение

```
        byte[] b1     = {1, 2, 3};
```

При инициализации элементов массива для элементов блока инициализации используются правила преобразования типов константных выражений, позволяющие, например, преобразовывать константное выражение типа `int` к типу `byte`, как это имеет место в приведенном примере. При передаче блока инициализации элементов массива в аргументе функционального члена, эти правила не работают, поэтому при трансляции приведенного ниже фрагмента программы будет выдано сообщение об ошибке, поскольку тип передаваемого массива не совпадает с типом формального параметра метода:

```
class Demo
{
        static void SomeFunc(short[] Array)
        {
                // Некоторая обработка
        }

        static void Main()
        {
                SomeFunc({3, 4, 5});
        }
}
```

Для устранения ошибки приведенный текст должен быть изменен следующим образом:

```
class Demo
{
        static void SomeFunc(short[] Array)
        {
                // Некоторая обработка
        }

        static void Main()
        {
                SomeFunc(new short[] {3, 4, 5});
        }
}
```

Преобразование типов переменных

В C# различают два вида преобразований типов переменных — *явное* и *неявное*. Неявное преобразование типов используется в том случае, если данное преобразование является естественным. Это относится, например, к преобразованию величины типа `float` в `double`. Такое преобразование является естественным, поскольку оба типа используются для представления значений с плавающей точкой и тип `double` имеет больший диапазон представления и большую точность, чем тип `float`. Поэтому при таком преобразовании не происходит потеря информации. При проведении неявного преобразования нет необходимости указывать тип, к которому производится преобразование.

При явном преобразовании необходимо в явном виде указать тип, к которому преобразуется тип исходной переменной. Этот тип должен иметь неявное преобразование к типу переменной, в которую будет записан результат преобразования. Необходимость указания типа, к которому производится преобразование, связано с тем, что неявное преобразование типов может быть выполнено различными способами, выбор которых может повлиять на результат преобразования. Если в приведенном ниже примере вместо явного преобразования попытаться использовать неявное, компилятор выдаст сообщение об ошибке:

```
using System;
class Test
{
        static void Main()
        {
                float  f = 1.23;
```

```
            double d = f; // Неявное преобразование
            f = (float) d;// Явное преобразование
        }
}
```

В C# все типы, включая типы значений, могут рассматриваться как объекты, поскольку все они являются потомками типа `object`. Поэтому методы объекта могут вызываться для переменных любого типа. Ниже приведен пример вызова метода `ToString` для целочисленной константы:

```
using System;
class Demo
{
        static void Main()
        {
            Console.WriteLine(17.ToString());
        }
}
```

В тех случаях, когда требуется использовать ссылку на переменную, имеющую тип значения, эта переменная копируется в объект, то есть присваивается переменной типа `object`. Эта операция называется *упаковкой значения* (*boxing*), а обратная ей операция — *распаковкой значения* (*unboxing*). Ниже приведен пример использования этих операций:

```
class Demo
{
        static void Main()
        {
            byte   i      = 17;
            object obj    = i;          // Копирование в объект
            byte   j      = (byte) obj; // Извлечение из объекта
        }
}
```

При распаковке значения может быть использовано только явное преобразование типов (для опытного программиста эти операции очевидны).

Переменные и формальные параметры

Для хранения и передачи значений в C# используются переменные и формальные параметры. *Переменная* представляет собой область памяти, выделенную для хранения некоторой информации. Каждая переменная имеет тип, определяющий формат и назначение хранящейся в ней информации. Среди

всех переменных выделяют особую группу, называемую *локальными переменными*. К локальным переменным относят переменные, объявленные в методах и в процедурах доступа к свойствам и индексаторам. Объявление локальных переменных, как правило, состоит из идентификатора типа переменной, идентификатора самой переменной и, возможно, из инициализатора данной переменной, присваивающего ей начальное значение. Например:

```
real   r = 1.0;
int    b;
```

Объявления нескольких переменных одного типа могут быть объединены в одну строку. Например:

```
       float  r = 1123, s, t = (float)1.0;
```

При инициализации переменной `t` использовано явное преобразование константы, поскольку действительные литералы в C# преобразуются к типу `double`. При инициализации переменной `r` использовано неявное преобразование типа `int`, к которому преобразуются целочисленные литералы, к типу `float`.

Переменная должна быть инициализирована перед первым своим использованием. То есть ей должно быть присвоено значение. В приведенном ниже примере содержится ошибка, поскольку в нем используется неинициализированная переменная:

```
class Demo
{
       void Some_Func()
       {
              int i;
              int j = 1;
              int k = i + j;// Ошибка! Переменная i
                                   // не инициализирована
       }
}
```

Другой разновидностью переменных являются поля, представляющие собой переменные, являющиеся членами класса или структуры. Если поле имеет модификатор `static`, оно называется *статическим полем*. Статические поля принадлежат всему классу, а не отдельному его объекту, поэтому при внесении изменений в это поле в одном объекте данного класса его значение изменяется для всех остальных его объектов. Если поле не имеет модификатора `static`, оно называется *нестатическим*. В приведенном ниже примере определены статическое и нестатическое поля:

```
class Some_Class
{
```

```
        private static bool   isEmpty;       // Статическое поле
        public string         Name;          // Нестатическое поле
}
```

Формальные параметры используются для передачи исходной информации методам и процедурам доступа, а также для получения от них результатов обработки. Различают четыре типа формальных параметров:

- передаваемые по значению;
- передаваемые по ссылке;
- выходные формальные параметры;
- массивы формальных параметров.

Формальные параметры, передаваемые по значению, используются для передачи методам и процедурам доступа значений некоторых переменных. При объявлении формальных параметров не указываются никакие модификаторы. Для хранения значений методы и процедуры доступа используют внутреннюю локальную переменную. Поэтому все изменения, которым подвергается значение в методе или процедуре доступа никак не отражается на значении исходной переменной. Это позволяет передавать в данном формальном параметре константы и помещать в соответствующий ему аргумент литералы. Поскольку в приведенном ниже примере формальный параметр передается по значению, при вызове данного метода использован целочисленный литерал:

```
class Some_Class
{
        void Some_Func( int i )
        {
                i = 0;
        }

        void Other_Func()
        {
                Some_Func( 1 );
        }
}
```

Формальные параметры, передаваемые по ссылке, имеют модификатор `ref`. В этом параметре передается не само значение, а ссылка на него. Все изменения данного параметра в методе или процедуре доступа немедленно отражаются на значении переданного параметра, поскольку для их хранения используется одна и та же область памяти. Тот факт, что значения формальных параметров, передаваемых по ссылке, могут изменяться, исключает

возможность передачи в них констант и литералов. Переменная, передаваемая в таком формальном параметре, должна быть инициализирована до первого вызова данного метода.

Ниже приведен пример использования формального параметра, передаваемого по ссылке. После вызова метода `Some_Func` переменная `k` принимает нулевое значение:

```
class Some_Class
{
        void Some_Func( ref int i )
        {
                i = 0;
        }

        void Other_Func()
        {
                int k = 1;
                Some_Func( ref k );
        }
}
```

Как видно из приведенного примера, модификатор `ref` должен использоваться как при объявлении метода, так и при обращении к нему. Использование этого ключевого слова при вызове метода повышает читабельность программ, указывая на то, что данный аргумент может изменять свое значение после завершения работы метода.

Выходные формальные параметры во многом аналогичны формальным параметрам, передаваемым по ссылке, однако, для них не требуется предварительная инициализация, поскольку эта процедура должна производиться в самом методе или процедуре доступа. Поэтому начальное значение выходного формального параметра, если он инициализирован, игнорируется. При объявлении и использовании выходных формальных параметров используется модификатор `out`.

В приведенном выше примере использования формального параметра, передаваемого по ссылке, отсутствует необходимость в предварительной инициализации переменной `k`. Поэтому данный формальный параметр может быть преобразован в выходной формальный параметр, как это показано ниже:

```
class Some_Class
{
        void Some_Func(out int i )
        {
```

```
            i = 0;
        }

        void Other_Func()
        {
            int k;
            Some_Func(out k );
        }
}
```

Все перечисленные типы формальных параметров имели прямое соответствие с аргументами, передаваемыми данному методу или процедуре доступа. То есть каждому формальному параметру соответствовал свой аргумент. Использование массивов формальных параметров позволяет передавать методу или процедуре доступа произвольное число формальных параметров. При этом все дополнительные формальные параметры помещаются в одномерный массив.

Для объявления массива формальных параметров используется модификатор `params`. Чтобы однозначно определить размер передаваемого массива, метод или процедура доступа должны иметь не более одного массива формальных параметров, который должен располагаться после всех остальных формальных параметров. Таким образом, членами этого массива становятся все аргументы, расположенные после аргументов, соответствующих другим формальным параметрам. Все элементы массива должны иметь один и тот же тип и передаваться по ссылке, то есть в них сохраняются изменения, внесенные в метод или в процедуру доступа. В данном массиве может не передаваться ни один аргумент. Рассмотрим это на примере:

```
class Some_Class
{
        void Some_Func(params int[] arr )
        {
            for(int i = 0; i < arr.Length; i++)
                arr[i] = 0;
        }

        static void Main()
        {
            int[]          arr    = {0, 1, 2, 3};
            Some_Class     sc     = new Some_Class();
```

```
            sc.Some_Func( );
            sc.Some_Func( 1, 2, 3 );
            sc.Some_Func(new int[] {1, 2, 3, 4});
            sc.Some_Func( arr );

            for(int i=0; i < arr.Length; i++ )
            {
                    Console.WriteLine( "arr[{0}] = {1}", i, arr[i] );
            }
        }
}
```

В результате выполнения приведенной программы на экран будут выведены нулевые значения элементов массива. Как видно из примера, при использовании массива формальных параметров нет необходимости передавать его размер, поскольку он может быть получен из самого объекта массива.

Выражения

Выражением называется последовательность операций, операндов и знаков препинания, задающих определенное вычисление. Операции будут рассмотрены в следующем разделе. Если в выражении используется несколько операций, то порядок их выполнения определяется приоритетом используемых операций. Все операции в языке программирования C# разбиты на группы приоритетов. В каждой из групп все операции имеют одинаковый приоритет. В основу иерархии приоритетов положено старшинство операций, принятое в математике. Так, например, операция умножения имеет больший приоритет, чем операция сложения и, поэтому, выполняется в первую очередь.

Если в выражении присутствуют две операции, имеющие одинаковый приоритет, то в силу вступает порядок их выполнения. Все бинарные операции, кроме операции присваивания, выполняются слева направо. То есть, последовательность операций `a*b*c` соответствует последовательности операций `(a*b)*c`.

Операция присваивания и условная операция (`?:`) выполняются справа налево. То есть, последовательность операций `a=b=c` соответствует последовательности операций `a=(b=c)`.

Для внесения изменений в приоритет и порядок выполнения операций используются скобки, представляющие собой операции наивысшего приоритета. Так в выражении `a+b*c` сначала выполняется операция умножения, а затем — операция сложения, а в выражении `(a+b)*c` — наоборот.

Операторы

C# унаследовал большинство своих операторов от языков C и C++, но внес в некоторые из них свои изменения и добавления.

Последовательности операторов и блоки

Последовательность операторов представляет собой один или несколько операторов, записанных последовательно. Если этого требует синтаксис, операторы в последовательности разделяются точкой с запятой.

Блоки представляют собой *составной оператор*. Необходимость в образовании составных операторов связана с тем, что практически каждый оператор можно рассматривать как набор ключевых слов и инструкций. Ключевые слова определяют последовательность действий, выполняемых оператором, а *инструкции* — выполняемые при этом действия. В качестве инструкции может выступать только один оператор. Однако в большинстве случаев простой оператор не может выполнить все действия, требуемые от него исходным оператором. Поэтому в качестве инструкций чаще всего используются составные операторы. Составной оператор представляет собой последовательность операторов, заключенную в фигурные скобки (`{}`).

Метки и оператор goto

Метки, установленные перед оператором, позволяют перейти к нему из любого места программы с использованием оператора `goto`. В качестве метки может выступать любой идентификатор (даже используемый в программе в той же области видимости), за которым следует двоеточие. Область действия метки ограничена блоком, в котором она определена (включая все вложенные блоки). Приведенный ниже пример выполняет те же операции, что и первое демонстрационное приложение, созданное в главе 2:

```
namespace Hello
{
using System;

class Hello
{
        static void Main()
        {
                goto   B;
                E: Console.WriteLine("I'm C#");
```

```
                return;
                B: Console.WriteLine("Hello, ");
                goto E;
        }
}
}
```

См. главу 2.

Условный оператор

Условный оператор позволяет выбрать исполняемый оператор в зависимости от значения оцениваемой в нем логической переменной. Существует две версии данного оператора. В первой из них определяется необходимость выполнения некоторого оператора без предоставления альтернативы, а во второй — производится выбор одной из альтернатив. Второй вариант отличается от первого наличием ключевого слова `else`. В приведенном ниже примере используется второй вариант условного оператора:

```
using System;
class Demo
{
        static void Main(bool Arg)
        {
                if ( Arg )
                        Console.WriteLine("False argument");
                else
                        Console.WriteLine("True argument");
        }
}
```

Оператор-переключатель

Оператор `switch` позволяет передавать управление одному из нескольких операторов с меткой варианта в зависимости от значения переключающего выражения. Для выхода из оператора `switch` после выполнения оператора, которому было передано управление, используется оператор `break`. При отсутствии оператора `break` исполнение программы будет продолжено со следующего оператора с меткой варианта:

```
using System;
class Demo
{
```

```
        static void Main(int Arg)
        {
                switch ( Arg)
                {
                        case 0:         Console.WriteLine("Argument is 0");
                                        break;
                        case 1:         Console.WriteLine("Argument is 1");
                                        break;
                default:        Console.WriteLine("Argument is {0}", Arg);
                                        break;
                }
        }
}
```

Оператор while

Оператор `while` циклически выполняет свою инструкцию до тех пор, пока проверяемое в нем логическое условие не примет значения `true`. Пример его использования приведен ниже:

```
using System;
class Demo
{
        static void Main()
        {
                int     i = 0;

                while( i < 10 )
                        Console.WriteLine( i++ );
        }
}
```

Оператор do

Оператор `do` отличается от оператора `while` тем, что проверка условия производится в нем не до, а после выполнения инструкции. Ниже приведена модификация рассмотренного выше примера, в котором вместо оператора `while` используется оператор `do`:

```
using System;
class Demo
{
```

```
        static void Main()
        {
                int    i = 0;

                do
                {
                        Console.WriteLine( i++ );
                }
                while( i < 10 )
        }
}
```

Оператор for

Оператор `for` выполняет следующую последовательность действий:

1. Вычисляет выражение инициализации. В этом выражении обычно инициализируются один или несколько счетчиков цикла.
2. Проверяется логическое условие. Если оно ложно (имеет значение `false`), управление передается следующему оператору.
3. Если условие истинно (имеет значение `true`), выполняется инструкция данного оператора.
4. Выполняется приращение одного или нескольких счетчиков цикла (или выполняется любая другая операция).
5. Производится переход к шагу 2.

Ниже приведен пример использования оператора `for`:

```
using System;
class Demo
{
        static void Main()
        {
                for( int i = 0; i < 10; i++)
                        Console.WriteLine( i );
        }
}
```

Оператор foreach

Оператор `foreach` просматривает содержимое коллекции, применяя инструкцию к каждому ее элементу:

```
using System;
using System.Collections;
```

```
class Demo
{
        static void PrintList(ArrayList Input)
        {
                foreach (object i in Input )
                        Console.WriteLine(i);
        }

        static void Main()
        {
                ArrayList temp = new ArrayList();

                for (int i = 0; i < 10; i++)
                        temp.Add(i);

                PrintList( temp );
        }
}
```

Оператор return

Оператор `return` возвращает управление методу, вызвавшему функцию, в которой расположен данный оператор. Оператор `return` должен содержать выражение того же типа, что и возвращаемое значение функции, в которой он расположен. Это выражение определяет значение, возвращаемое функцией вызвавшему ее методу. Если функция не возвращает значения (имеет тип возвращаемого значения `void`), то оператор `return` не содержит никакого выражения.

Операторы throw и try

Операторы `throw` и `try` используются для работы с исключениями. Первый из них вызывает исключение, а второй — их обрабатывает. Оператор `try` предоставляет возможность задания блока, который будет выполняться при любом выходе из блока оператора `try`.

Операторы checked и unchecked

Операторы `checked` и `unchecked` используются для задания режима проверки переполнения при выполнении арифметических операций и операций преобразования типа над целочисленными операндами. При возникновении переполнения в результате выполнения операции, расположенной в операторе

checked, в зависимости от ее операндов компилятор выдает сообщение об ошибке, или в процессе выполнения приложения вызывается исключение OverflowException. Если же переполнение возникает в результате выполнения операции, расположенной в операторе unchecked, никакое сообщение об ошибке не выдается, а в полученном результате отбрасываются старшие биты, которые не могут быть сохранены в формате результата данной операции.

Оператор lock

Оператор lock используется для обеспечения взаимно исключающего доступа к объекту. Этот оператор получает доступ к разделяемому объекту, выполняет свой вложенный оператор и освобождает разделяемый объект.

Классы

Классы используются для создания новых типов ссылок. C# не поддерживает множественного наследования классов, но поддерживает множественное наследование интерфейсов. Кроме того, в классе может быть реализовано несколько интерфейсов.

Членами класса могут быть константы, поля, методы, свойства, индексаторы, события, перегруженные операции, конструкторы, деструкторы и объявления вложенных типов. Каждый член класса имеет свой модификатор режима доступа. Имеется пять режимов доступа:

- public — член класса доступен в любом контексте;
- protected — член класса доступен только в данном классе и во всех производных от него классах;
- internal — член класса доступен только в пределах своей сборки (как правило, в пределах глобального пространства имен данного приложения);
- protected internal — член класса доступен только в данном классе и во всех производных от него классах, принадлежащих той же сборке;
- private — член класса доступен только в данном классе.

Константы

Константа представляет собой член класса, содержащий постоянное значение, вычисляемое компилятором. Значения констант могут вычисляться на основании значений других констант, если при этом не образуются циклические зависимости. Поскольку значения констант определяются в процессе компиляции, до создания любого объекта класса, они являются статическими, и их полное имя содержит имя класса, а не объекта.

```
class Some_Class
{
        public const int      i = 1;
        public const int      j = i + 1;

        static void Main()
        {
                Console.WriteLine("i = {0}, j = {1}", Some_Class.i,
Some_Class.j);
        }
}
```

Поля

Поле представляет собой переменную, являющуюся членом класса или структуры. Поля могут быть статическими и нестатическими. Статические поля принадлежат классу, и изменение значения поля в любом объекте данного класса отражается на всех других объектах этого класса. Для обозначения статических полей используется модификатор `static`. Нестатические поля принадлежат объектам класса, и изменение их значений в одном объекте никак не отражается на значении этого поля в других объектах данного класса. В определении нестатического поля отсутствует модификатор `static`.

Статические поля могут использоваться для создания внутренних констант класса, значения которых определяются при его инициализации, то есть, в процессе выполнения приложения. Для того, чтобы значения этих переменных не могли изменяться, при их объявлении используется модификатор `readonly`. Значение такому полю может присваиваться только при его объявлении или в конструкторе класса.

В приведенном ниже примере рассмотрен класс, используемый для работы с точкой на плоскости. В этом классе задана внутренняя константа для обозначения нулевой точки:

```
class My_Point
{
        internal      int    x;
        internal      int    y;

        public My_Point (int x, int y)
        {
                this.x = x;
                this.y = y;
        }
```

```
        public static readonly My_Point            Null_Point = new
        My_Point(0, 0);
}
```

Методы

Методы представляют собой члены класса, выполняющие некоторые преобразования для данного класса или его объекта. Методы имеют список формальных параметров, который может быть пуст, и возвращают значение. Если в качестве типа возвращаемого значения метода указан тип `void`, то этот метод не возвращает никакого значения. Методы могут быть статическими и нестатическими. Как и в случае полей, при объявлении статического метода указывается модификатор `static`. Статические методы вызываются для всего класса, поэтому их полное имя включает в себя имя класса. Нестатические методы вызываются для конкретного объекта класса, поэтому их полное имя включает идентификатор объекта или ключевое слово `this`, если они вызываются в пределах своего класса.

В приведенном ниже примере продемонстрировано использование статических и нестатических методов:

```
class Some_Class
{
        void Some_Func(int i )
        {
                ...
        }

        static void Other_Func()
        {
                ...
        }

        void Third_Func()
        {
                Some_Class     sc = new Some_Class();

                this.Some_Func( 1 );          // Нестатический метод
                sc.Some_Func( 1 );            // Нестатический метод
                Some_Class.Other_Func();      // Статический метод
        }
}
```

Методы могут перегружаться, то есть несколько методов могут иметь одно и то же имя, но разный набор формальных параметров. Два набора формальных параметров считается разными, если у них не совпадает число формальных параметров или тип хотя бы одного из формальных параметров. Два метода не могут отличатюся только типом возвращаемого значения. Продемонстрируем перегрузку функций, внеся изменения в предыдущий пример:

```
class Some_Class
{
        void Some_Func(int i )
        {
                ...
        }

        static void Some_Func()
        {
                ...
        }

        void Third_Func()
        {
                Some_Class     sc = new Some_Class();

                this.Some_Func( 1 );          // Нестатический метод
                sc.Some_Func( 1 );           // Нестатический метод
                Some_Class.Some_Func();      // Статический метод
        }
}
```

Свойства

Свойства представляют собой именованные атрибуты, связанные с классом или его объектом. Примером свойства являются длина строки, размер шрифта, заголовок окна, имя пользователя и т. д. Свойства так же, как и поля, имеют имя, тип и одинаковый синтаксис доступа. Однако для хранения свойств не выделяется память. Для доступа к свойству используются специальные процедуры доступа (*accessors*), определяющие выражение, которое необходимо исполнить при чтении или записи значений свойств. Таким образом, свойства предоставляют возможность чтения, записи и вычисления значения атрибутов объекта.

Свойства так же, как и поля, имеют тип, и синтаксис доступа к ним не отличается от синтаксиса доступа к полям. Для чтения и записи значений свойств используются специальные *процедуры доступа*. Объявление свойства можно

разбить на две части, первая из которых аналогична объявлению поля, а вторая — содержит описание процедур доступа к данному свойству. В приведенном ниже примере в классе `Some_Class` объявлено свойство `My_Property`:

```
class Some_Class
{
        privateint    i;
        public int    My_Property
        {
                get
                {
                        return i;
                }
                set
                {
                        i = value;
                }
        }
}
```

Свойство может иметь только процедуру доступа по чтению (`get`), процедуру доступа по записи (`set`) или обе эти процедуры. В первом случае свойство доступно только для чтения, во втором случае — только для записи, а в третьем случае — как для чтения, так и для записи. Как правило, процедура доступа по чтению возвращает значение некоторого закрытого поля класса, а процедура доступа по записи записывает в него значение предопределенного формального параметра `value`. Ниже приведен пример использования определенного выше свойства:

```
static void Main()
{
        Some_Class    sc = new Some_Class();
        sc.My_Property= 1;   // get
        sc.My_Property++;    // get set
        Console.WriteLine("Property = {0}", sc.My_Property);     // set
}
```

События

Событие используется объектом или классом для посылки извещения. Для этого в классе объявляется событие. Объявление события имеет форму объявления поля или свойства, в которое добавлено ключевое слово `event`. Объявляемое поле или свойство должно иметь тип делегата. Как правило, события объявляются как поля. Однако в тех случаях, когда число событий

велико и большинство из них не будут вызываться, поля лучше определять как свойства, поскольку в этом случае экономится оперативная память. Примером редко используемых событий являются сообщения об ошибках, число которых намного превышает число выдаваемых сообщений в реальном приложении.

C# предоставляет единое решение для работы с событиями, позволяющими классам определить извещения, для которых в приемнике может быть создан исполняемый код в виде обработчика события. События являются одним из основных способов обращения к библиотекам классов.

Ниже приведен пример объявления события в классе:

```
public delegate void EventHandler(object sender, EventArgs e);
public class Demo
{
        public event EventHandler My_Event;
        protected void OnEvent(EventArgs e)
        {
                if (My_Event != null)
                        My_Event(this, e);
        }
        public void Reset()
        {
                My_Event = null;
        }
}
```

Тип делегата `EventHandler` является членом пространства имен `System` и может не указываться. В данном примере член пространства имен `System` скрывается, и вместо него используется полностью эквивалентный ему член текущего пространства имен.

В пределах класса, в котором оно объявлено, событие может рассматриваться как закрытое (`private`) поле, имеющее тип делегата (в данном случае — `EventHandler`). За пределами этого класса событие также рассматривается как поле, однако, оно может использоваться только в левой части операции `+=` или операции `-=`. Операция `+=` добавляет обработчик события, а операция `-=` удаляет его. Ниже приведен пример использования описанного выше класса `Demo`:

```
public class Use_Demo
{
        Demo   First;
        public Use_Demo ()
        {
```

```
                First = new Demo();
                First.My_Event += new EventHandler(My_Handler);
        }
        public void Detach()
        {
                First.My_Event -= new EventHandler(My_Handler);
                First = null;
        }
        void My_Handler(object sender, EventArgs e)
        {
                // Обработка события First.My_Event
        }
}
```

В приведенном примере обработчик события `My_Haydler` подключается к событию `My_Event` в конструкторе класса `Use_Demo` и отключается от него в методе `Detach` того же класса.

В некоторых случаях события могут объявляться не как поля, а как свойства класса. При таком объявлении класс `Demo` будет выглядеть следующим образом:

```
public class Demo
{
        public event EventHandler My_Event
        {
                get { ... }
                set { ... }
        }
        public void Reset()
        {
                My_Event = null;
        }
}
```

Операции

Операции используются для определения операций, производимых над объектами данного класса. Различают унарные, бинарные, тернарные операции и операции преобразования типов.

Ниже определен пользовательский целочисленный класс, осуществляющий проверку на переполнение:

```
public struct integer
{
```

```
        short value;
        public integer(int value)
        {
                if (value < -32768.0 || value > 32767.0 )
                        throw new ArgumentException();
                this.value = (short) value;
        }
        public static implicit operator short(integer i)  { return
i.value; }
        public static implicit operator integer(int s)           { return
new integer(s); }
}
class Test
{
        static void Main()
        {
                integer a = 5;
                integer b = 3;
                integer c = a + b;
                c = a*b;
        }
}
```

В рассмотренном примере создан класс, хранящий значение типа `short` и проверяющий на переполнение все осуществляемые с ним операции. Проверка диапазона производится в конструкторе данного класса. Поскольку в нем определены только операции преобразования, то все остальные операции над объектами данного класса производятся путем преобразования хранимого значения к типу `short`, выполнения требуемой операции и преобразования результата к типу `integer`. Обе операции преобразования имеют модификатор `implicit`, то есть являются неявными операциями преобразования типов.

Индексаторы

Индексаторы позволяют обращаться к объекту класса как к массиву. Объявление индексатора аналогично объявлению свойства за тем исключением, что индексатор не имеет имени (вместо имени используется ключевое слово `this`, за которым в квадратных скобках располагаются дополнительные аргументы индексатора). Пример:

```
public class Demo
{
```

```
        private int[] int_array;
        public string this[int index]
        {
                get
                {
                        return int_array[index];
                }
                set
                {
                        int_array[index] = value;
                }
        }
}
```

В данном примере индексатор использован для доступа к закрытому массиву `int_array`. Этот индексатор используется следующим образом:

```
static void Main()
{
        Demo   d = new Demo();
        d[1]   = 10;                // Процедура set
        Console.WriteLine( d[1] );  // Процедура get
}
```

Конструкторы объектов класса

Конструкторы объектов класса используются для инициализации его объектов. Если в классе не определен ни один конструктор его объектов, то в него автоматически добавляется конструктор, не имеющий аргументов и вызывающий для всех членов класса, требующих инициализации, их конструкторы, используемые по умолчанию.

В приведенном ниже примере в классе `Some_Class` определены два конструктора его объектов. Первый из этих конструкторов не имеет аргументов и эквивалентен конструктору, который был бы добавлен в данный класс по умолчанию. Второй конструктор производит инициализацию полей класса значениями своих аргументов:

```
class Some_Class
{
        privateint    i;
        public int    j;
        public Some_Class()
        {
```

```
            i = 0;
            j = 0;
        }
        public Some_Class( int i, int j)
        {
            this.i = i;
            this.j = i;
        }
}
```

Деструкторы

Деструктор класса используется для корректного завершения работы объекта класса и освобождения выделенной ему памяти. Деструктор не может иметь аргументов, при его объявлении не могут использоваться модификаторы доступа, он не может вызываться в явном виде. Деструктор объекта класса вызывается сборщиком мусора. Поскольку освобождать выделенную память приходится только при работе с *небезопасным* (*unsafe*) кодом, то в большинстве случаев используются деструкторы классов, помещаемые в них по умолчанию.

Статические конструкторы класса

Статический конструктор осуществляет инициализацию статических членов класса. Иногда говорят, что он инициализирует сам класс до инициализации его объектов. Так же, как и деструктор, статический конструктор не может иметь аргументов, при его объявлении не могут использоваться модификаторы доступа, он не может вызываться в явном виде. Статический конструктор класса автоматически вызывается при загрузке класса. Необходимость объявления в классе статического конструктора возникает в тех случаях, когда для инициализации статических членов класса не могут быть использованы конструкторы, вызываемые по умолчанию.

Наследование

Классы не поддерживают множественного наследования, то есть каждый класс может иметь не более одного базового класса. Если для класса не указан его базовый класс, то он является непосредственным потомком класса `object`.

В приведенном ниже примере класс `Derived_Class` является непосредственным потомком класса `Base_Class`:

```
class Base_Class
{
```

```
        public void Some_Func()      { ... }
}
class Derived_Class: Base_Class
{
        public void Other_Func() { ... }
}
class Demo
{
        static void Main()
        {
                Derived_Class  d = new Derived_Class();
                d.Some_Func();        // Унаследовано от Base_Class
                d.Other_Func();              // Определено в Derived_Class
                Base_Class    b = d; // Инициализация
                b.Some_Func();
        }
}
```

Как видно из приведенного выше примера, производный класс имеет доступ к открытым (`public`) членам базового класса, а объект базового класса может инициализироваться объектом производного класса.

Методы, свойства и индексаторы класса могут быть виртуальными, то есть их реализация может быть перегружена в производных классах. Для доступа к перегруженному члену базового класса может использоваться ключевое слово `base`, как это показано в приведенном ниже примере:

```
class Base_Class
{
        public virtual void Some_Func()     { ... }
}
class Derived_Class: Base_Class
{
        public override void Some_Func()
        {
                base. Some_Func();
                ...
        }
}
class Demo
{
```

```
        static void Main()
        {
                Derived_Class  d = new Derived_Class();
                d.Some_Func();        // Вызов Derived_Class.Some_Func
                Base_Class     b = d; // Инициализация
                b.Some_Func();        // Вызов Derived_Class.Some_Func
                b = new Base_Class(); // Инициализация
                b.Some_Func();        // Вызов Base_Class.Some_Func

        }
}
```

В рассмотренном примере два раза вызывается метод производного класса и только один раз — метод базового класса. Хотя второй вызов осуществляется из объекта базового класса, этот объект является копией объекта производного класса, в котором данный метод базового класса перегружен.

C# позволяет объявлять абстрактные классы, в объявлении которых используется модификатор `abstract`. Эти классы являются незавершенными, поэтому невозможно создать их объекты. В абстрактных классах могут быть объявлены абстрактные функциональные члены, то есть функциональные члены, не имеющие реализации. Для того, чтобы класс, производный от абстрактного, перестал быть абстрактным, в нем необходимо перегрузить все абстрактные члены базового класса. Поскольку абстрактные члены класса обязательно должны быть перегружены, то они по умолчанию являются виртуальными, и перегружающие их члены производного класса должны иметь модификатор `override`. Ниже приведен пример создания неабстрактного класса, производного от абстрактного класса:

```
abstract class Base_Class
{
        public abstract void Some_Func();
}
class Derived_Class: Base_Class
{
        public override void Some_Func()
        {
                ...
        }
}
```

Структуры

Структуры могут содержать некоторые типы членов классов и могут реализовывать интерфейсы. Однако межу структурами и классами существуют существенные различия:

- структуры являются типом значения, а классы — типом ссылки;
- структуры не поддерживают наследования;
- значения структур хранятся в стеке или являются внедренными в программу.

При написании быстродействующих приложений необходимо учитывать особенности использования классов и структур. Например, массивы структур создаются быстрее и доступ к ним проще, чем к массивам классов. Это связано с тем, что для каждого объекта класса необходимо выделять отдельную область памяти и для обращения к ней использовать ссылку. Объекты структуры создаются в стеке, и для нескольких объектов может быть выделена одна область памяти, для обращения к которой используется одна ссылка, а для обращения к элементу массива может быть использована адресная арифметика. С другой стороны, объекты классов передаются по ссылке, а объекты структур — по значению. Поэтому в качестве аргументов функциональных членов лучше использовать классы.

В качестве примера рассмотрим реализацию объекта для хранения цвета в виде объект класса и ост класса:

```
class Color_Class
{
        public byte   Red;
        public byte   Green;
        public byte   Blue;

        public Color_Class (byte Red, byte Green, byte Blue)
        {
                this.Red              = Red;
                this.Green     = Green;
                this.Blue      = Blue;
        }
}
struct Color_Struct
{
        public byte   Red;
        public byte   Green;
        public byte   Blue;
```

```
        public Color_Struct (byte Red, byte Green, byte Blue)
        {
                this.Red              = Red;
                this.Green    = Green;
                this.Blue     = Blue;
        }
}
```

Как видно из приведенного примера, объявления класса и структуры практически не отличаются.

Интерфейсы

Интерфейс представляет собой абстрактный тип, содержащий объявления методов, свойств, события и индексаторов. Эти объявления используются для создания некоторого единого интерфейса между различными приложениями, реализация которого полностью ложится на плечи реализующего его приложения. Ниже приведен пример объявления интерфейса, содержащего члены всех вышеперечисленных типов:

```
interface IDemo
{
        bool   this[int index] { get; set; }
        event  Event_Handler E;
        short  Some_Func(short s);
        int    Some_Properety { get; set; }
}
public delegate void Event_Handler(object sender, Event e);
```

Интерфейсы поддерживают множественное наследование. То есть один интерфейс может быть непосредственным потомком нескольких интерфейсов.

Члены интерфейсов реализуются в классах и структурах. При этом в одном классе или структуре может быть одновременно реализовано несколько интерфейсов. Интерфейсы могут быть реализованы двумя способами. При использовании первого способа реализации в классе или структуре, реализующей данный интерфейс, объявляется и реализуется соответствующий член, имеющий модификатор права доступа `public`. При применении второго способа реализации члена интерфейса необходимо использовать его полное имя. Этот способ называется явной реализацией члена интерфейса, поскольку в нем в явном виде указывается, метод какого интерфейса в данном случае реализуется. При явной реализации члена интерфейса его вызов производится с использованием имени или объекта интерфейса, а не того

класса или структуры, в котором он был реализован. В этом случае нет необходимости использовать модификатор права доступа. Ниже приведен пример использования обоих способов реализации:

```
class Demo
{
        interface IDemo
        {
                void   First_Func(int i );
                void   Second_Func(int i);
        }
        class CDemo : IDemo
        {
                int    i;
                public void   First_Func(int i )    { this.i = i;}
                void   IDemo.Second_Func(int i)     { this.i = i;}
        }

        static void Main()
        {
                CDemo  cdemo = new CDemo();
                cdemo.First_Func( 1 );
                cdemo.Second_Func( 1 );      // Ошибка

                IDemo idemo = cdemo;
                idemo.First_Func( 1 );
                idemo.Second_Func( 1 );
        }
}
```

Данная программа содержит ошибку, поскольку в классе `CDemo` отсутствует реализация метода `Second_Func`.

Делегаты

Делегаты используются в тех случаях, когда C++ и другие алгоритмические языки используют указатели на функции. В отличие от указателей на функции делегаты являются объектно-ориентированными, безопасными и сохраняющими тип.

Все типы делегатов являются потомками базового класса `System.Delegate`. Объект делегата инкапсулирует метод. Если этот метод не является статиче-

ским, этот экземпляр также содержит ссылку на объект, для которого вызывается данный метод, и ссылку на сам метод. Для вызова такого метода необходимо указать объект делегата и соответствующий набор аргументов. Если метод является статическим, то делегат хранит только ссылку на сам метод.

Поскольку при вызове делегата указывается объект, для которого вызывается данный метод, то ему безразлично, какому классу он принадлежит. Единственное, что его интересует, это соответствие передаваемого списка аргументов его списку формальных параметров. Это делает делегаты мощным средством для "анонимного" вызова.

Прежде чем использовать объект делегата, необходимо его не только объявить, но и создать его объект. Для объявления делегата используется специальный синтаксис. Например, делегат `My_Delegate`, не имеющий списка формальных параметров и возвращающий значение `void`, объявляется следующим образом:

```
delegate void My_Delegate();
```

Для создания объекта делегата используется ключевое слово `new`, а аргументом конструктора делегата является ссылка на статический или нестатический метод, имеющий тот же набор формальных параметров и тип возвращаемого значения, что и делегат. После создания объекта делегата он может использоваться для вызова методов. Рассмотрим пример использования объявленного выше делегата:

```
class Demo
{
        static void Some_Func()
        {
                // Некоторая обработка
        }
        static void Main()
        {
                My_Delegate md = new My_Delegate (Some_Func);
                md();
        }
}
```

В результате выполнения этой программы будет вызван статический метод `Some_Func`. Вообще-то, в данном контексте проще было бы сразу вызвать этот метод. Однако тип делегата может быть использован как тип одного из аргументов функционального члена. В этом случае одна и та же функция может производить различную обработку в зависимости от того, какой аргумент типа делегата будет ей передан. Этот аргумент может, например, задавать поведение функции в нештатных ситуациях. Таким образом, одна и та

же функция может разным образом реагировать на эти ситуации в зависимости от контекста ее вызова, определяющего требования к надежности получаемого результата. Например:

```
void Other_Func(int i, My_Delegate error)
{
        ...
        if( i < 0 )
        {
                error();
        }
        ...
}
```

Перечислимые типы

Перечислимые типы представляют собой набор символьных констант, определяющий все возможные значения данного типа. Перечислимые типы используются, как правило, для облегчения работы с наборами различных параметров. Например, вместо использования порядковых номеров дней недели можно создать перечислимый тип из семи элементов:

```
enum Week {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
           Sunday}
```

Использование этого перечислимого типа не только облегчает чтение программы, в которой он определен, но и снимает неопределенность, связанную с нумерацией дней недели в России и на Западе. Мы привыкли, что неделя начинается с понедельника, что естественно для христиан, а на западе, испытывающем иудейское влияние, первым днем недели считается воскресенье, последним и, как следует из Библии, выходным — суббота. Таким образом, номера дней недели могут быть восприняты неоднозначно, в то время как символьная константа `Sunday` всегда будет обозначать воскресенье, независимо от того, соответствует ли ей значение `0` или значение `6`.

Пространства имен и сборки

Все представленные до сих пор примеры программ были относительно невелики и не общались с другими программами и библиотеками, за исключением используемого в них класса `System.Console`, определенного в системной библиотеке. Большинство реальных программ состоит из нескольких взаимодействующих компонентов. Некоторые из этих компонентов могут быть разработаны специально для конкретной системы, а другие — закуплены у поставщиков программного обеспечения.

Для обеспечения корректной работы приложений, компоненты которых разрабатывались независимо друг от друга, в C# используются *пространства имен* и *сборки*. Пространства имен используются как для внутренней организации программы, так и для внешней организации системы, то есть для упорядочения способа предоставления своих элементов другим программам.

Сборки используются для объединения компонентов системы при ее распространении. Сборка представляет собой контейнер типов, содержащий исполняемые коды, реализующие некоторые из этих типов и ссылки на другие сборки, содержащие реализацию оставшихся типов. Различают два типа сборок: приложения и библиотеки. Приложения имеют главную входную точку и их файлы, как правило, имеют расширение .exe. Библиотеки не имеют одной входной точки и их файлы, как правило, имеют расширение .dll.

Пространства имен имеют иерархическую структуру, то есть одно пространство имен может включать в себя одно или несколько других пространств имен. Microsoft рекомендует использовать в качестве идентификатора внешнего пространства имен имя компании-разработчика. В него должны быть включены тематические пространства имен, определяющие основные компоненты создаваемого приложения. Принцип выбора идентификаторов для пространств имен более низкого уровня отдается на откуп разработчику.

Поскольку элементы, принадлежащие одному пространству имен, могут располагаться в различных частях программы, и даже в различных файлах, в C# предоставлена возможность одновременного задания сразу нескольких пространств имен. Например, фрагмент программы

```
namespace bhv
{
        namespace CSharp
        {
                namespace Demo
                {
                        public class Test
                        {
                                ...
                        }
                }
        }
}
```

может быть записан в виде

```
namespace bhv.Csharp.namespace.Demo
{
```

```
        public class Test
        {
                ...
        }
}
```

При создании приложения, использующего класс `Test`, необходимо указать его полное имя `bhv.CSharp.namespace.Demo.Test`. Это слишком длинное имя для практического применения, тем более что оно должно использоваться для доступа к членам данного класса, которые могут выступать в качестве элементов различных выражений. Для упрощения доступа к этому классу используется директива `using`, позволяющая осуществлять доступ ко всем элементам данного пространства имен как к элементам данного пространства имен. Например:

```
using bhv.Csharp.namespace.Demo;
class New_Class
{
        static void Main()
        {
                Test   t = new Test();
                ...
        }
}
```

Другим способом сократить длину идентификаторов является использование альтернативных имен. Этот способ может использоваться, например, в том случае, когда в текущем пространстве имен уже определен элемент с тем же именем, что и элемент внешнего пространства имен. Например:

```
using Old_Test = bhv.Csharp.namespace.Demo.Test;
public class Test
{
        ...
}
class New_Class
{
        static void Main()
        {
                Old_Test      t = new Old_Test();
                ...
        }
}
```

Работа с версиями

Работа с версиями продукта в большинстве языков не поддерживается, однако, это не относится к C#. Под работой с версиями понимают или обеспечение совместимости на уровне исходных кодов с предыдущими версиями, когда для обеспечения работоспособности приложений, работавших со старой версией продукта, достаточно перекомпилировать их исходные коды с новой версией продукта, или совместимость на уровне двоичных кодов, когда для обеспечения работоспособности приложений, работавших со старой версией продукта, нет необходимости производить их перекомпиляцию.

Большинство языков не поддерживают совместимость на уровне двоичных кодов, и большинство из них в очень ограниченном объеме поддерживают совместимость с исходными кодами. Некоторые языки программирования содержат ограничения, вообще не позволяющие вносить изменения в класс, не внося одновременно с этим изменений в приложения приемников.

Как пример, рассмотрим класс `Base`, содержащийся в некоторой библиотеке классов. В первой версии данной библиотеки класс `Base` не содержит метода `Some_Func`. Исходя из этого пользователь данной библиотеки создал класс `Derived`, производный от класса `Base`, включил в него метод `Some_Func` и широко использовал этот класс в своих разработках.

```
// Библиотека классов
namespace First
{
        class Base    // Первая версия
        {
                // Реализация класса
        }
}
// Пользователь
namespace Second
{
        class Derived: First.Base
        {
                public virtual void Some_Func()
                {
                        // Реализация метода
                }
        }
}
```

До сих пор все нормально. Однако предположим, что во второй версии класса Base появился метод Some_Func:

```
// Библиотека классов
namespace First
{
        class Base     // Вторая версия
        {
                // Реализация класса
                public virtual void Some_Func()
                {
                        // Реализация метода
                }
        }
}
```

Вторая версия класса Base должна быть совместима с первой его версией как на уровне исходных кодов, так и на уровне двоичных кодов. Однако новый метод Some_Func вступает в конфликт с одноименным методом, объявленным в классе Derived. Как должен разрешиться этот конфликт? Если метод Derived.Some_Func перегружает метод Base.Some_Func, то возникает вопрос о соответствии метода Derived.Some_Func лицензии на использование метода Base.Some_Func. В этой лицензии может быть, например, указано, что метод базового класса обязательно должен вызываться в перегруженном методе. Естественно, что такое требование невыполнимо.

Данные конфликты имен нельзя рассматривать как редкие исключения из правил, поскольку авторы библиотеки классов и ее пользователи работают абсолютно независимо и контакты между ними в большинстве случаев исключены. Как правило, классы библиотеки интенсивно используются, и каждый из них служит базовым для множества производных классов, образующих сложные иерархии. Конфликты имен чаще всего возникают при работе со специализированными классами, ориентированными на работу в некоторой узкой области. В этой области сформировалась своя терминология, и каждый из авторов попытается использовать ее для своих имен, что неизбежно приведет к конфликтам, поскольку для обозначения одного понятия используется крайне ограниченный набор терминов.

Для разрешения конфликтов версий в C# от разработчиков требуется в явном виде указать свои намерения. В исходном коде конфликт отсутствует: поскольку класс Base не содержит метода Some_Func, этот метод в производном классе Derived является новым, а не перегружает несуществующий метод базового класса.

Если в новой версии класса `Base` появляется метод `Some_Func`, то одноименный метод класса `Derived` не имеет к нему никакого отношения. Новая версия класса `Base` предоставляет новые функциональные возможности, о которых не знал, а, соответственно, и не использовал, разработчик класса `Derived`. Поэтому, даже при работе с новой версией библиотеки, метод `Some_Func` класса `Derived` следует рассматривать как новый метод, а не как перегрузку метода базового класса.

При перекомпиляции приложения, использующего библиотеку классов, возможны два случая: пользователь библиотеки не знает о существовании или не хочет использовать метод `Base.Some_Func`, или же он знает о нем и включил в реализацию своего метода `Derived.Some_Func`. Поскольку возможны оба варианта, компилятор C# по умолчанию скрывает метод базового класса и выдает предупреждение. Таким образом, он восстанавливает ситуацию, существовавшую до перекомпиляции, и сообщает пользователю о неиспользованных им возможностях, предоставляемых новой версией библиотеки. Если перегрузка метода базового класса не представляется целесообразной, пользователь библиотеки явным образом сообщает об этом компилятору с использованием ключевого слова `new` при объявлении своего метода. Использование этого ключевого слова подавляет выдачу предупреждения, не изменяя результатов компиляции:

```
// Библиотека классов
namespace First
{
        class Base     // Вторая версия
        {
                // Реализация класса
                public virtual void Some_Func()
                {
                        // Реализация метода
                }
        }
}

// Пользователь
namespace Second
{
        class Derived: First.Base
        {
                new public virtual void Some_Func()
                {
```

```
                        // Реализация метода
                }
        }
}
```

Если же метод базового класса предоставляет пользователю дополнительные возможности, делающие целесообразной перегрузку метода базового класса вместо использования старого метода, пользователь библиотеки должен явным образом указать на это, используя ключевое слово `override`:

```
// Библиотека классов
namespace First
{
        class Base     // Вторая версия
        {
                // Реализация класса
                public virtual void Some_Func()
                {
                        // Реализация метода
                }
        }
}

// Пользователь
namespace Second
{
        class Derived: First.Base
        {
                new public virtual void Some_Func()
                {
                        base.Some_Func();
                        // Дополнительные действия
                }
        }
}
```

Если возможности, предоставляемые методом базового класса, должны быть использованы в производном классе, но его одноименный метод выполняет другие действия, то этот метод может быть переименован. Однако это приведет к тому, что этот класс станет несовместимым со своими потомками, что, впрочем, не имеет особого значения в том случае, если производный класс не используется другими программами.

Атрибуты

Как и все остальные процедурные языки программирования, C# использует декларативные элементы. Примером таких декларативных элементов являются модификаторы. Однако стандартные модификаторы не могут полностью удовлетворить потребность в декларативных элементах. Во многих случаях пользователю необходимо связать с некоторым элементом языка свой декларативный элемент, формат которого не может быть определен заранее. Для создания таких декларативных элементов и применяются атрибуты.

Атрибуты используются для хранения дополнительной информации о программных единицах, которая может быть получена в процессе выполнения приложения. Для получения информации из атрибутов в процессе выполнения приложения используется механизм отражения. Разделяют зарезервированные и пользовательские атрибуты. Зарезервированные атрибуты используются компилятором, и пользователь не может взять их имена для своих атрибутов.

Например, программист может создать атрибут `AuthorAttribute` и поместить в него имя разработчика данной программной единицы и ее версию. Для этого ему нужно создать специальный класс атрибута, производный от класса `System.Attribute`, как это показано в приведенном ниже примере:

```
[AttributeUsage(AttributeTargets.All)]
public class AuthorAttribute: System.Attribute
{
        string szAutor;
        public AuthorAttribute(string author);
        public string Query { get { ... } }
        public double Version;
}
```

Поскольку имена всех атрибутов должны заканчиваться словом `Attribute`, то это слово может опускаться и имя атрибута `AuthorAttribute` будет выглядеть как имя `Author`. Если при объявлении класса атрибута в его имени отсутствует окончание `Attribute`, то оно подставляется автоматически и приведенное выше правило замены имен продолжает действовать.

При указании атрибута ему передаются позиционные и именованные параметры. В качестве позиционных параметров выступают формальные параметры конструктора класса атрибута. Если класс атрибута имеет несколько конструкторов, то набор позиционных параметров, используемый при указании атрибута, должен соответствовать набору формальных параметров одного из конструкторов класса атрибута. Таким образом, набор позиционных параметров определяет вызываемый конструктор класса атрибута. Позиционные параметры являются обязательными, и ни один из них не может быть опущен. Атрибут имеет `AuthorAttribute` один позиционный параметр

типа string. В качестве именованных параметров выступают его открытые (public) нестатические поля или свойства, доступные как для чтения, так и для записи. Атрибут AuthorAttribute имеет один именованный параметр Version. Свойство Query не является именованным параметром данного атрибута, поскольку оно используется только для чтения.

Рассмотрим пример использования данного атрибута:

```
[Author("John Bool", Version=2.7)]
class Some_Class
{
       // Реализация класса
}
```

Для получения хранящейся в атрибуте информации в процессе выполнения приложения используется механизм отражения NGWS. В приведенном ниже примере из класса Some_Class извлекается информация об авторе и версии:

```
using System;
class Demo
{
       static void Main()
       {
          Type type = typeof(Some_Class);
          object[] arr = type.GetCustomAttributes(typeof(AuthorAttribute));
          if (arr.Length == 0)
              Console.WriteLine("Class Some_Class has no Help attribute.");
          else
          {
          for (int i=0; i<arr.Length; ++i)
          {
                if (arr[i] is License)
                {
                Author a = (Author)arr[i];
                Console.WriteLine( "Author = {0} Version {1}", a.Query,
                // l.Version );
                      }
              }
              }
       }
}
```

В приведенном примере с использованием метода `GetCustomAttributes` получается массив атрибутов данного класса, затем в этом массиве производится поиск атрибута заданного типа и только после этого выводится содержащаяся в нем информация.

Исключения

C# использует структурированный, унифицированный и проверяющий соответствие типов подход для обработки ошибок, как на системном уровне, так и на уровне приложения. Этот подход реализован в форме обработки *исключений*. Механизм обработки исключений C# во многом аналогичен механизму обработки исключений C++, однако, имеются и некоторые различия:

- если в C++ в качестве исключения могло использоваться любое значение любого типа, то в C# все исключения являются объектами класса, производного от класса `System.Exception`;
- при обработке исключений в C# может быть использован блок `finally`, который будет выполнен независимо от того, было ли вызвано исключение в блоке `try` или этот блок нормально завершил свою работу. Написание подобного кода в C++ потребует введения значительной избыточности и не гарантирует идентичность обработки, поскольку отсутствует механизм обеспечения идентичности соответствующих блоков;
- в C# определены классы исключений, вызываемые при возникновении переполнения, деления на ноль и нулевой ссылки, и обеспечен механизм вызова их в приложениях.

Как уже говорилось выше, базовым классом для всех типов исключений является класс `System.Exception`. Этот класс содержит, помимо прочего, два доступных только для чтения свойства, к которым можно обратиться из любого типа исключения. Значения этих свойств могут быть заданы в конструкторе класса `System.Exception`.

- Свойство `Message` содержит текстуальное описание причины вызова исключения.
- Свойство `InnerException` содержит "внутреннее исключение" для данного исключения. Если это поле не имеет значения `null`, то данное исключение было вызвано при обработке другого исключения. Доступ к этому исключению обеспечивается свойством `InnerException`.

Исключения могут быть вызваны двумя способами:

- для немедленного и безусловного вызова исключения используется оператор `throw`. Этот оператор относится к операторам перехода, поэтому его конечная точка недостижима;
- исключения могут вызываться и при выполнении некоторых операторов или выражений C#, если выполнение этих операторов или вычисление

значения выражений не может быть нормально завершено. Например, если при делении целых чисел знаменатель равен нулю, вызывается исключение `System.DivideByZeroException`.

Обработка исключений производится в операторе `try`. При вызове исключения в блоке `try` система производит поиск ближайшего блока `catch`, обрабатывающего данный тип исключений. Сначала поиск производится в том операторе `try`, в котором было вызвано исключение. При этом все обработчики `catch` просматриваются в порядке их расположения в операторе `try`. Если этот поиск завершился безрезультатно, поиск производится в операторе `try`, включающем в себя текущий оператор `try`. Если метод не содержит внешних операторов `try`, поиск производится в методе, вызвавшем его. При этом полагается, что вложенный оператор `try` располагается в месте вызова данного метода. Считается, что обработчик `catch` обрабатывает данный тип исключения, если тип обрабатываемого исключения совпадает с типом вызванного исключения или является его базовым типом. Если в обработчике `catch` не указан тип исключения, то он обрабатывает исключение типа `System.Exception`, то есть исключения любого типа.

Если в результате выполнения описанной выше процедуры был найден обработчик исключения, то управление передается данному обработчику. Если в операторе `try` не было обнаружено нужного обработчика `catch`, то перед тем, как передать управление внешнему оператору `try`, управление передается блоку `finally` данного оператора, если таковой присутствует.

Если обработчик вызванного типа исключения не был найден, то:

- если поиск привел в статический конструктор или инициализатор статического поля, в достигнутой точке статического конструктора или инициализатора вызывается исключение `System.TypeInitializationException`. Свойство `InnerException` данного исключения возвращает исходное вызванное исключение;
- если поиск привел к программному коду, инициирующему поток или программу, выполнение данного потока или программы завершается.

Глава 4

Лексическая структура языка

Программа C# состоит из одного и более исходных файлов. Исходные файлы C# представляют собой упорядоченную последовательность символов Unicode. Обычно эти файлы соответствуют файлам используемой пользователем файловой системы, но это соответствие нигде не требуется.

Компиляция программы производится в три этапа:

1. Предварительная обработка, позволяющая включить в текст программы внешние фрагменты текста или исключить из нее некоторые фрагменты текста.
2. Лексический анализ, преобразующий текст исходных файлов в последовательность взаимосвязанных лексических единиц.
3. Синтаксический анализ, преобразующий полученную последовательность лексических единиц в исполняемый код программы.

Директивы препроцессора

Предварительная обработка текста программы основана на анализе директив препроцессора, указывающих источники включаемых фрагментов и выделяющие фрагменты, которые должны быть исключены из окончательного текста программы.

По сравнению с языками C и C++ возможности препроцессора C# существенно урезаны. В нем, например, отсутствует директива `#include`, использовавшаяся для включения файлов в текст других файлов и, следовательно, возможность такого действия. Исключение этой директивы существенно ускоряет процесс компиляции приложения, поскольку при его отсутствии создается более строгая структура приложения. Кроме того, отсутствие этой директивы заставило отказаться от файлов заголовков, что привело к более тесной связи объявления и реализации объектов (теперь это практически одно и то же).

Директива препроцессора может представлять собой:

- набор идентификаторов, ключевых слов, литералов, операций и знаков препинания, за которыми должен располагаться символ новой строки. В качестве символа новой строки могут выступать:
  - символ возврата каретки (`U+000D`);

- символ перевода строки (U+000A);
- оба эти символа;
- символ разделителя строк (U+2028);
- символ разделителя абзацев (U+2029).

❒ определение идентификаторов;
❒ условная секция;
❒ управляющая строка;
❒ номер строки.

Определение идентификаторов

Определение идентификаторов препроцессора производится с использованием директивы #define, за которой следует объявляемый идентификатор. Эта директива объявляет идентификатор для всего файла, в котором он расположен. Обратная операция выполняется директивой #undef. Если идентификатор, указанный в директиве #undef, был до этого определен директивой #define, он становится неопределенным. Определение идентификатора можно рассматривать как связывание с ним значения true. Можно также считать, что с неопределенным идентификатором связано значение false. В качестве определяемого идентификатора может использоваться любой идентификатор или ключевое слово, кроме true и false.

> ***Примечание***
> Директиву #define не следует путать с одноименным макросом, использовавшимся в языках C и C++ для макрозамены текста (например, для переименования типов). Отказ от этого макроса существенно повысил читабельность программы, поскольку теперь любое незнакомое имя можно найти в одном из пространств имен, видимых в данном контексте.

Определенность идентификаторов проверяется условными директивами, которые будут рассмотрены в следующем разделе. В результате использования этих директив программный текст

```
#defineFirst
#undef Second
class Some_Class
{
#if First
       void First_Func() {}
#else
       void Second_Func() {}
```

```
#endif
#if Second
       void Third_Func() {}
#else
       void Fourth_Func() {}
#endif
}
```

будет преобразован препроцессором к виду:

```
class Some_Class
{
       void First_Func() {}
       void Fourth_Func() {}
}
```

Определение идентификатора (то есть директивы `#define` и `#undef`) должно предшествовать любым элементам языка, не являющимся директивами препроцессора. Например, не содержит ошибки следующий фрагмент программы:

```
#define First
#if First
       #define Second
#endif
namespace Global
{
class Some_Class
{
#if Second
       void Third_Func() {}
#else
       void Fourth_Func() {}
#endif
}
}
```

Однако фрагмент, приведенный ниже, содержит ошибку.

```
class Some_Class
{
#define First // Ошибка!
#if First
       void First_Func() {}
#else
```

```
        void Second_Func() {}
#endif
}
```

Казалось бы, проверка определенности идентификатора, определенного в предыдущей строке, не имеет никакого смысла, однако, не следует забывать, что эти строки могут находиться в разных файлах, поскольку проверка определенности идентификатора может производиться в первой строке включаемого файла.

Директивы `#define` и `#undef` могут неоднократно использоваться для одного и того же идентификатора, однако, определение уже определенного идентификатора и отмена определения неопределенного идентификатора не приводят ни к каким изменениям в работе препроцессора.

Условная секция

Условная секция, пример которой был приведен в предыдущем разделе, состоит из директивы `#if` и следующего за ним выражения, после которого должен располагаться символ новой строки. Синтаксис языка C# позволяет размещать за этой строкой любые элементы языка, в том числе и директивы препроцессора, что несколько затрудняет стройное изложение материала.

Из всех возможных директив препроцессора, которые могут быть помещены за строкой директивы `#if`, следует выделить директивы `#elif` и `#else`. Они могут находиться только между директивами `#if` и `#endif`.

Директива `#elif` представляет, по сути, комбинацию директив `#else #if`, между которыми отсутствуют другие директивы. Поскольку директива `#else` должна соответствовать последней директиве `#elif` или исходной директиве `#if`, она должна располагаться в конце группы директивы `#if`, и в этой группе может присутствовать только одна такая директива. Синтаксис строки директивы `#elif` совпадает с синтаксисом строки директивы `#if`, в котором директива `#if` заменена директивой `#elif`. Строка директивы `#else` состоит из самой директивы и символа новой строки. За этими строками могут располагаться любые элементы языка C#. При этом следует помнить, что указание директивы препроцессора `#endif` обозначает конец условной секции. За директивой `#endif` должен располагаться символ новой строки.

Управляющие строки препроцессора

Для обеспечения возможности создания программистом собственных сообщений об ошибках и предупреждений компилятора в C# используются директивы `#error` и `#warning`. После этих директив могут помещаться текстовые сообщения, которые будут выводиться, когда компилятор дойдет до соответствующей строки программы. В приведенном ниже примере выдает-

ся предупреждение о том, что этот код очень опасен, а в том случае, если окажутся одновременно определенными идентификаторы `MICROSOFT` и `SUN`, будет выдано сообщение невозможности совместной работы программного обеспечения Microsoft и Sun:

```
#warning Very dangerous code
#define MICROSOFT
#if MICROSOFT && SUN
       #error It would not work!
#endif
class Some_Class
{
}
```

Изменение номера строки

Для изменения номера строки и имени исходного файла, используемого компилятором при выдаче предупреждений и сообщений об ошибках, разработчик программного обеспечения может использовать директиву `#line`. Если эта директива не используется, компилятор автоматически определяет указываемые в сообщениях номер строки и имя файла. Потребность в такой директиве возникает в связи с возможностью включения текста одного файла в текст другого , что может привести к ошибочным сообщениям компилятора. За директивой `#line` обязательно указывается целочисленный литерал, содержащий номер строки. За целочисленным литералом может располагаться строковый литерал, содержащий имя файла. Поскольку строковый литерал заключается в парные кавычки " (`U+0022`), то он не может содержать данный символ. Кроме того, этот строковый литерал не может содержать пробелов (хотя длинные имена файлов могут содержать пробелы).

Выражения препроцессора

Идентификаторы препроцессора могут объединяться в выражения операциями `!`, `==`, `!=`, `&&` и `||`. Эти операции полностью идентичны соответствующим логическим операциям и поэтому здесь подробно рассматриваться не будут. Кроме того, для группировки выражений препроцессора могут использоваться круглые скобки.

Все директивы препроцессора должны начинаться с начала строки. То есть перед ними не должно быть пробелов.

Комментарии могут располагаться после всех директив препроцессора, за исключением его управляющих строк. Это связано с тем, что любой текст,

расположенный после директив `#error` и `#warning` рассматривается как текст соответствующего сообщения. Например, строка

```
#define Debug // Определение идентификатора препроцессора
```

будет рассматриваться как строка, содержащая комментарий, а строка

```
#error Fatal error  // Сообщение об ошибке
```

выдаст сообщение об ошибке "Fatal error // Сообщение об ошибке".

Регулярные комментарии не могут начинаться или заканчиваться на той же строке, что и директива препроцессора. В приведенном ниже примере ошибка будет выдана для каждой директивы препроцессора:

```
/* Это неправильный комментарий, поскольку он заканчивается
на одной строке с директивой препроцессора */ #define First
/* Этот комментарий расположен на одной строке с директивой препроцессо-
ра*/ #define Second
#define Third /* Этот комментарий расположен на одной строке с директивой
препроцессора*/
#define Fourth /* Это неправильный комментарий, поскольку он начинается
на одной строке с директивой препроцессора */
```

Лексический анализ

Как уже говорилось выше, в результате лексического анализа текст исходных файлов преобразуется в последовательность взаимосвязанных лексических единиц. При этом из текста исходных файлов прежде всего исключаются пробелы и комментарии. После этого в нем выделяются следующие лексические единицы:

- ❒ идентификаторы;
- ❒ ключевые слова;
- ❒ литералы;
- ❒ операции и знаки препинания.

Пробелы игнорируются только в том случае, если они не служат для разделения отдельных лексических единиц. При лексическом анализе в качестве пробелов рассматриваются следующие символы:

- ❒ символ новой строки, в качестве которого может выступать:
  - символ возврата каретки (`U+000D`);
  - символ перевода строки (`U+000A`);
  - оба эти символа;
  - символ разделителя строк (`U+2028`);
  - символ разделителя абзацев (`U+2029`).
- ❒ символ вертикальной табуляции (`U+0009`);

- символ подачи бланка (`U+000C`) (form feed);
- символ "подстановки" (`U+001A`) (**<Ctrl>+<Z>**);
- любой символ класса `Zs` Unicode.

В C# используются два различных типа комментариев. Регулярные комментарии начинаются последовательностью символов `/*` и заканчиваются последовательностью символов `*/`. Подобные комментарии могут занимать часть строки, всю строку или несколько строк. Регулярные комментарии не могут содержать комбинацию символов `*/`. Ниже приведен пример использования регулярных комментариев:

```
/*
       Демонстрационное приложение
*/
class Some_Class      /* Класс демонстрационного приложения */
{

       static void Main()
       {
              /* Реализация главного метода демонстрационного приложения
*/
       }
}
```

Строковый комментарий начинается с последовательности символов `//` и заканчивается с концом строки. Например:

```
//
//     Демонстрационное приложение
//
class Some_Class      // Класс демонстрационного приложения
{

       static void Main()
       {
              // Реализация главного метода демонстрационного приложения
       }
}
```

Синтаксический анализ

Синтаксический анализ преобразует последовательность лексических единиц, полученную в результате проведения лексического анализа, в исполняемый код программы.

Идентификаторы

Идентификатором называется имя, используемое для обозначения программного элемента (поля, метода, свойства и т. д.). Приведенные ниже правила именования идентификаторов соответствуют требованиям стандарта Unicode 3.0, технический акт 15, приложение 7. Единственными отклонениями от этого стандарта являются:

- допущение использования символов подчеркивания и аналогичных ему символов в качестве первого символа идентификатора;
- допущение использования в идентификаторах эскейп-последовательностей Unicode;
- использование символа `@` для превращения ключевых слов в идентификаторы.

В качестве идентификаторов не могут использоваться ключевые слова языка C#. Для того, чтобы ключевое слово языка C# могло бы использоваться в качестве идентификатора, перед ним без какого-либо разделителя ставится символ `@`.

В качестве первого символа идентификатора может использоваться:

- символы Unicode, принадлежащие классам `Lu`, `Ll`, `Lt`, `Lm`, `Lo` или `Nl` (буквы);
- символы Unicode, принадлежащие классу `Pc` (связующие символы);
- эскейп-последовательности Unicode, представляющие символы классов `Lu`, `Ll`, `Lt`, `Lm`, `Lo`, или `Nl` (буквы);
- эскейп-последовательности Unicode, представляющие символы класса `Pc` (связующие символы).

В качестве остальных символов идентификатора могут использоваться:

- символы Unicode, принадлежащие классам `Lu`, `Ll`, `Lt`, `Lm`, `Lo` или `Nl` (буквы);
- символы Unicode, принадлежащие классу `Nd` (десятичные цифры);
- символы Unicode, принадлежащие классу `Pc` (связующие символы);
- символы Unicode, принадлежащие классам `Mn` или `Mc` (комбинирующие символы);
- символы Unicode, принадлежащие классу `Cf` (символы форматирования);
- эскейп-последовательности Unicode, представляющие символы классов `Lu`, `Ll`, `Lt`, `Lm`, `Lo`, или `Nl` (буквы);
- эскейп-последовательности Unicode, представляющие символы класса `Nd` (десятичные цифры);
- эскейп-последовательности Unicode, представляющие символы класса `Pc` (связующие символы);

- эскейп-последовательности Unicode, представляющие символы классов `Mn` или `Mc` (комбинирующие символы);
- эскейп-последовательности Unicode, представляющие символы класса `Cf` (символы форматирования).

Примерами идентификаторов являются `i`, `name1`, `@do`.

Префикс `@` используется для преобразования ключевых слов в идентификаторы. Это преобразование бывает полезным при взаимодействии с программами, написанными на других языках программирования. Поскольку символ `@` не является частью идентификатора, то в других языках программирования это ключевое слово будет использоваться как обычный идентификатор без префикса. Символ `@` может ставиться и перед обычными идентификаторами, но использование этого символа в подобном контексте настоятельно не рекомендуется, поскольку существенно снижает читабельность программы. В приведенном ниже примере формальным параметром метода является целочисленный идентификатор `int`:

```
int Some_Func ( int @int )
{
       return @int++;
}
```

Два идентификатора считаются одинаковыми, если они совпадают после проведения следующих операций:

- из них удаляется префикс `@`;
- любая эскейп-последовательность Unicode преобразуется в соответствующий символ Unicode;
- производится нормализация идентификатора в соответствии с формой нормализации C (технический акт 15 консорциума Unicode).

Ключевые слова

Ключевые слова представляют собой идентификаторы, зарезервированные языком программирования для своих внутренних нужд. Они не могут использоваться в качестве пользовательского идентификатора без префикса `@`.

В качестве ключевых слов зарезервированы следующие идентификаторы:

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal

is	lock	long	namespace	new	null
object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackallo c	static	string
struct	switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe	ushort
using	virtual	void	while		

Литералы

Литерал используется для представления в исходном коде различных значений.

Разделяют следующие виды литералов:

- логический литерал;
- целочисленный литерал;
- литерал формата с плавающей точкой;
- символьный литерал;
- строковый литерал;
- нулевой литерал.

Логический литерал

Логический литерал преобразуется к типу `bool` и может принимать два значения — `true` и `false`.

Целочисленный литерал

Целочисленный литерал может быть преобразован к одному из следующих типов: `int`, `uint`, `long` или `ulong`. Тип, к которому преобразуется литерал, определяется его значением и наличием или отсутствием суффикса типа. Целочисленный литерал может быть десятичным и шестнадцатеричным. Десятичный целочисленный литерал представляет собой последовательность десятичных чисел, за которой может располагаться суффикс типа. Шестнадцатеричный целочисленный литерал представляет собой последовательность десятичных чисел или символов `A`, `B`, `C`, `D`, `E`, `F`, `a`, `b`, `c`, `d`, `e` и `f`, за которой может располагаться суффикс типа. В качестве суффиксов типа могут использоваться следующие комбинации символов: `U`, `u`, `L`, `l`, `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU` и `lu`.

Примерами целочисленных литералов являются:

```
123
-789
0xffff
```

Тип целочисленного литерала определяется следующим образом:

- если литерал не имеет суффикса типа, то его типом является первый тип в следующей последовательности, в который он может быть преобразован: `int`, `uint`, `long`, `ulong`;
- если литерал имеет суффикс `U` или `u`, то его типом является первый тип в следующей последовательности, в который он может быть преобразован: `uint`, `ulong`;
- если литерал имеет суффикс `L` или `l`, то его типом является первый тип в следующей последовательности, в который он может быть преобразован: `long`, `ulong`;
- если литерал имеет суффиксы `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU` или `lu`, то он имеет тип `ulong`.

  Если значение литерала находится за пределами диапазона величин, имеющих тип `ulong`, компилятор выдает сообщение об ошибке.

В общем случае литералы не включают в себя знак числа, однако имеются два исключения, связанные с тем, что диапазон отрицательных чисел на единицу превышает диапазон положительных чисел. Поэтому:

- Если перед десятичным целочисленным литералом 2147483648 · (2^{31}) стоит знак унарной операции `-`, этот литерал имеет тип `int` и значение -2147483648 · (-2^{31}). Во всех остальных случаях этот целочисленный литерал будет иметь тип `uint`.
- Если перед десятичным целочисленным литералом 9223372036854775808 · (2^{63}) стоит знак унарной операции -, этот литерал имеет тип `long` и значение –9223372036854775808 · (-2^{63}). Во всех остальных случаях этот целочисленный литерал будет иметь тип `ulong`.

Литерал формата с плавающей точкой

Литерал формата с плавающей точкой может быть преобразован к одному из следующих типов: `float`, `double` или `decimal`. Тип, к которому преобразуется литерал, определяется его значением и наличием или отсутствием суффикса типа. Литерал формата с плавающей точкой может состоять из двух наборов десятичных цифр, разделенных точкой, набора десятичных цифр, перед которым стоит точка, или целочисленного десятичного литерала. Две последние формы литерала формата с плавающей точкой могут включать в себя порядок, состоящий из знака + или -, символа `E` или `e` и набора десятичных чисел. Последним элементом каждого литерала формата с плавающей точкой может быть суффикс типа. В качестве суффиксов типа могут использоваться следующие символы: `F`, `f`, `D`, `d`, `M` и `m`.

Если литерал не имеет суффикса типа, то он преобразуется к типу `double`. В противном случае тип литерала определяется его суффиксом типа:

- суффиксы типа `F` и `f` указывают на то, что данный литерал имеет тип `float`. Например, `5f`, `.1F` и `3.2e8f`;
- суффиксы типа `D` и `d` указывают на то, что данный литерал имеет тип `double`. Например, `3.62D`, `.3333e5d` и `3.1415926d`;
- суффиксы типа `M` и `m` указывают на то, что данный литерал имеет тип `decimal`. Например, `762.7M`, `12.3e15m` и `59M`;

Если литерал формата с плавающей точкой не может быть преобразован к одному из указанных выше типов, компилятор выдает сообщение об ошибке.

Символьный литерал

Символьные литералы преобразуются к типу `char`. Символьные литералы представляют собой символ, заключенный в одиночные кавычки. В качестве символов могут выступать любые символы, кроме одиночной кавычки (‘), обратной наклонной черты (\) и любого пробела, кроме простого пробела ( ), простые эскейп-последовательности, перечисленные в табл. 4.1, шестнадцатеричные эскейп-последовательности, представляющие собой эскейп-последовательность `\x`, за которой следует последовательность шестнадцатеричных цифр, и эскейп-последовательности Unicode.

Таблица 4.1. *Простые эскейп-последовательности*

Эскейп-последовательность	Наименование	Код Unicode.
\'	Апостроф	0x0027
\"	Кавычки	0x0022
\\	Обратная косая черта	0x005C
\0	Нуль-символ	0x0000
\a	Звуковой сигнал	0x0007
\b	Возврат на шаг (Backspace)	0x0008
\f	Перевод страницы	0x000C
\n	Перевод строки	0x000A
\r	Возврат каретки	0x000D
\t	Горизонтальная табуляция	0x0009
\v	Вертикальная табуляция	0x000B

Строковый литерал

Строковые литералы преобразуются к типу `string`. В C# могут использоваться два типа строковых литералов: регулярные и дословные. Регулярные строковые литералы представляют собой последовательность символов и простых или шестнадцатеричных эскейп-последовательностей, заключенных в двойные кавычки. Регулярный строковый литерал может состоять только из двойных кавычек, в которые не заключен ни один символ или эскейп-последовательность. В этом случае считается, что данный литерал пуст. В регулярных строковых литералах действует то же ограничение на набор используемых символов, что и в символьных литералах.

Дословный строковый литерал состоит из символа `@`, за которым следует последовательность символов, заключенная в двойные кавычки. Основное отличие дословного строкового литерала от регулярного состоит в том, что в нем все символы воспринимаются "дословно", то есть не подвергаются преобразованию. Единственным исключением из этого правила являются двойные кавычки. Таким образом, подобные строковые литералы могут содержать фрагменты, совпадающие с простыми и шестнадцатеричными эскейп-последовательностями, которые не будут преобразовываться при работе с данным литералом. Дословные строковые литералы могут содержать несколько строк текста.

Ниже приведены примеры строковых литералов:

```
string first  = "Привет, я C#";                  // Привет, я C#
string second = @"Привет, я C#";                 // Привет, я C#
string third  = "Привет, \t я C#";               // Привет, я C#
string fourth = @"Привет, \t я C#";              // Привет, \t я C#
string fifth  = "Очередной \"баг\" в программе"; // Очередной ""баг"" в
                                                 //программе
string sixth  = @"Очередной ""баг"" в программе"; // Очередной ""баг"" в
                                                  // программе
string seventh = "\\\\server\\MyDoc\\Text.doc"; // \\server\MyDoc\Text.doc
string eighth = @"\\server\\MyDoc\\Text.doc"; // \\server\MyDoc\Text.doc
string ninth  = "Раз\nДва\nТри";
string tenth  = @"Раз
                Два
                Три";
```

Строковые литералы, присваиваемые переменным `ninth` и `tenth`, демонстрируют различные способы вывода трех строк текста.

Нулевой литерал

Нулевой литерал состоит из ключевого слова `null`. Этот литерал преобразуется к нулевому типу.

Операции и знаки препинания

Операции используются для создания выражений, содержащих один и более операнд. Операнды в выражениях объединяются знаками операций. Например, в выражении `i*j` операция `*` используется для перемножения двух операндов: `i` и `j`. Знаки препинания используются для группировки и разделения операторов и выражений. Например, знак препинания `;` используется для разделения выражений в списке выражений.

Для обозначения операций и знаков препинания используются знаки, приведенные в табл. 4.2.

Таблица 4.2. *Знаки для обозначения операций и знаков препинания*

{	}	[	]	(	)	.	,	:	;
+	-	*	/	%	&	|	^	!	~
=	<	>	?	++	--	&&	||	<<	>>
==	!=	<=	>=	+=	-=	*=	/=	%=	&=
|=	^=	<<=	>>=	->					

Эскейп-последовательность Unicode

Эскейп-последовательности Unicode используются для представления символов Unicode и могут использоваться в идентификаторах, а также в строковых и символьных литералах.

Эскейп-последовательность Unicode представляет собой последовательность из четырех шестнадцатеричных цифр, начинающуюся с последовательности символов `\u`. Каждая эскейп-последовательность соответствует одному символу Unicode, поэтому при необходимости задания с помощью эскейп-последовательностей нескольких символов для каждого из них должна быть указана своя эскейп-последовательность.

В приведенном ниже примере эскейп-последовательность `\u0022` используется для помещения выводимого текста "Test" в парные кавычки:

```
class Demo
{
        static void Main()
        {
                Console.WriteLine("\u0022 {0} \u0022", "Test" );
        }
}
```

Глава 5

Пространства имен

Базовым элементом программы на языке C# является пространство имен, которое может содержать *объявление типа* или вложенное пространство имен. Под объявлением типа понимается объявление класса, структуры, интерфейса, перечислимого типа или типа делегата. Элементы объявления типа зависят от объявляемого типа. Например, объявление класса может включать в себя статические и нестатические конструкторы, деструкторы, поля, константы, методы, свойства, индексаторы, события, объявления перегруженных операций и вложенных типов.

Объявление определяет имя в том пространстве имен, в котором оно было произведено. В одном и том же пространстве имен не может быть объявлено два одинаковых имени. Исключение сделано для перегружаемых конструкторов, методов, индексаторов и знаков перегружаемых операций. Однако даже в этом случае не допускается объявлять в пространстве имен два одинаковых имени для различных типов данных, например, для поля и свойства.

При работе с пространствами имен определены следующие правила:

- *Глобальное* пространство имен включает в себя все пространства имен, определенные во всех файлах программы, не включенные ни в одно другое пространство имен.
- Во всех файлах программы объявления, произведенные в пространствах имен, имеющих одно и то же полное имя, относятся к одному и тому же пространству имен.
- Каждое объявление нового класса, структуры или интерфейса представляет собой новое пространство имен. В этом пространстве имен не может быть объявлен член, имеющий то же имя, что и класс или структура, образующие данное пространство имен. Исключение сделано только для статических или нестатических конструкторов данного класса или структуры. В пространстве имен класса, структуры или интерфейса допускается перегрузка методов и индексаторов. В пространстве имен класса или структуры, кроме того, допускается перегрузка конструкторов и операций. Это означает, например, что класс, структура или интерфейс могут содержать объявления нескольких одноименных методов, различающихся своим списком формальных параметров (имеющих разные сигнатуры). Поскольку пространство имен базового класса или интерфейса не вклю-

чается в пространство имен производного класса или интерфейса, то в производном классе или интерфейсе может быть объявлен член с тем же именем, что и в базовом классе. При этом одноименный член базового класса скрывается независимо от того, принадлежит ли он к тому же типу членов типа (методов, индексаторов, свойств и т. д.), что и объявленный член, или нет.

- Объявление каждого перечислимого типа создает новое пространство имен.
- Каждый блок, включая блок оператора-переключателя, создает отдельное пространство имен для объявленных в нем локальных переменных. Если этот блок является телом метода или конструктора, то в его пространстве имен определены также все переменные из списка формальных параметров данного метода или конструктора. Пространство имен вложенного блока включает в себя пространство имен внешнего блока.
- Каждый блок, включая блок оператора-переключателя, создает отдельное пространство имен для меток. Поскольку пространство имен вложенного блока включает в себя пространство имен внешнего блока, во вложенном блоке не может быть определена метка с тем же именем, что и в любом из его внешних блоков.

Обычно порядок объявления в пространстве имен не имеет значения. Это относится, прежде всего, к пространствам имен, типам, константам, методам, свойствам, индексаторам, операциям, конструкторам и деструкторам. Порядок объявления важен только в следующих случаях:

- порядок объявления полей и локальных переменных определяет порядок их инициализации;
- локальные переменные должны быть определены до своего первого использования;
- порядок констант в объявлении перечислимого типа определяет сопоставляемые им значения, если эти значения не заданы явным образом.

Поскольку пространства имен с одним и тем же полным именем образуют одно пространство имен, в них не могут быть определены члены с одним и тем же именем. Например:

```
namespace bhv.Temp
{
        class Demo
        {
                ...
        }
}
namespace bhv.Temp
{
```

```
        class Demo    // Ошибка. Класс уже определен.
        {
                ...
        }
}
```

Эта ошибка возникнет даже в том случае, если объявления этих пространств имен будут располагаться в различных файлах.

В отличие от C и C++, в C# объявление переменной во вложенном блоке не скрывает переменной, определенной во внешнем блоке. Это правило действует даже в том случае, когда переменная во внешнем блоке определяется после вложенного блока, в котором была определена одноименная переменная, то есть, после того, как она была уничтожена при выходе из этого блока. Рассмотрим это на примере:

```
class Some_Class
{
        int Some_Func(bool b)
        {
                if ( b )
                {
                        int i = 1;
                        return i;
                }
                int    i = 2; // Ошибка
                return i;
        }
        int Other_Func(bool b)
        {
                if ( b )
                {
                        int i = 1;
                        return i;
                }
                else
                {
                        int    i = 2; // OK
                        return i;
                }
        }
}
```

Члены пространств имен и типов

Пространства имен и типы содержат члены. Для доступа к ним используются простые и полные имена. Простые имена используются в том случае, когда объект или тип, которому принадлежит тот или иной член, может быть определен из контекста. Если же контекст не позволяет однозначно идентифицировать член, используется полное имя, состоящее из идентификатора объекта или класса, которому принадлежит этот член, и простого его имени, разделяемых точкой.

Члены типа могут быть определены в нем или могут быть унаследованы им от своего базового класса. Из базового класса наследуются все члены, кроме его конструкторов и деструкторов, независимо от установленных для них режимов доступа. Однако не все унаследованные члены базового класса доступны в производном типе. Член базового класса может быть недоступен вследствие установленного для него режима доступа или потому, что он скрыт членом производного класса.

- Пространства имен и типы, не вложенные в другие пространства имен, принадлежат глобальному пространству имен. Они составляют имена, имеющие глобальную область видимости. Пространства имен и типы, определенные в других пространствах имен, являются членами данных пространств имен. Нельзя ограничить доступ к пространству имен, поэтому они не могут объявляться с модификаторами `private`, `protected` или `internal`. Они всегда имеют режим доступа `public`.
- Членами структуры являются члены, в ней определенные, и члены, унаследованные этой структурой от класса `object`. Примерами структуры C# являются простые типы значений.
- Членами перечислимого типа являются объявленные в данном типе константы и члены, унаследованные им от класса `object`.
- Членами класса являются члены, объявленные в данном классе, и члены, унаследованные им от его базовых классов (исключая класс `object`, который не является базовым классом). В классе могут быть определены константы, поля, методы, конструкторы, деструкторы, свойства, события, индексаторы, операции и определенные в нем вложенные типы. Наследуются все перечисленные выше члены класса, кроме его конструкторов и деструкторов. Наследование производится независимо от режима доступа к наследуемым членам базового класса.
- Членами интерфейсов являются члены, определенные в данном интерфейсе, а также члены, унаследованные им от всех своих базовых интерфейсов и класса `object`.
- Членами массива являются члены, унаследованные им от класса `System.Array`.
- Членами делегата являются члены, унаследованные им от класса `System.Delegate`.

Объявления членов позволяют устанавливать для них режим доступа. Режим доступа к члену определяется не только указанным в его объявлении режимом доступа, но и режимом доступа типа, в котором он объявлен. Если в данном контексте разрешен доступ к члену, то он является *доступным*. В противном случае он является *недоступным*. Член является доступным, если его вызов производится в пределах его *области действия*.

Установка режима доступа

Для членов может быть установлен один из следующих режимов доступа:

- *открытый*, устанавливается модификатором `public`. Доступ к члену не ограничен;
- *защищенный*, устанавливается модификатором `protected`. Доступ к члену может быть осуществлен только из класса, в котором он объявлен, и из типов, являющихся потомками данного класса;
- *локальный*, устанавливается модификатором `internal`. Доступ к члену может быть осуществлен только в проекте, в котором он объявлен;
- *локальный защищенный*, устанавливается модификатором `protected internal`. Доступ к члену может быть осуществлен только в проекте, в котором он объявлен, и только из класса, в котором он объявлен, и из типов, являющихся потомками данного класса;
- *закрытый*, устанавливается модификатором `private`. Доступ к члену может быть осуществлен только из класса, в котором он объявлен.

Режимы доступа не могут устанавливаться произвольно. Набор допустимых режимов доступа существенно зависит от контекста объявления. Если при объявлении какого-либо члена опущен модификатор режима доступа, то режим доступа к этому члену устанавливается, исходя из контекста его объявления:

- в объявлениях пространств имен не могут использоваться модификаторы режима доступа. По умолчанию все пространства имен имеют режим доступа `public`;
- типы, объявленные в глобальном пространстве имен или в других пространствах имен могут иметь модификаторы права доступа `public` или `internal`. По умолчанию для них устанавливается режим доступа `internal`;
- члены класса могут иметь любой из перечисленных выше режимов доступа. По умолчанию для них устанавливается режим доступа `private`;
- члены структуры могут иметь режимы доступа `public`, `internal` или `private`. По умолчанию для них устанавливается режим доступа `private`;

- поскольку члены интерфейса могут иметь только режим доступа `public`, при их объявлении не могут использоваться модификаторы режима доступа;
- поскольку члены перечислимого типа могут иметь только режим доступа `public`, при их объявлении также не могут использоваться модификаторы режима доступа.

Область действия членов

Областью действия членов называется фрагмент программы, в котором разрешен доступ к данному члену. Этот фрагмент программы необязательно должен образовывать связную область.

Для определения области действия введем следующие определения:

- член является членом верхнего уровня, если он не определен ни в одном другом типе;
- член является вложенным, если он определен в другом типе;
- под программным текстом проекта понимается текст всех программ, содержащихся во всех исходных файлах данного проекта;
- под программным текстом типа понимается весь программный текст, помещенный между открывающей и закрывающей фигурными скобками тела класса, структуры, интерфейса или перечислимого типа (включая все вложенные типы).

Область действия предопределенных типов неограничена.

Область действия типа верхнего уровня определяется следующим образом:

- если этот тип имеет открытый режим доступа (`public`), то его область действия распространяется на весь программный текст проекта, в котором он определен, и любого проекта, использующего данный проект;
- если этот тип имеет локальный режим доступа (`internal`), то его область действия распространяется только на программный текст проекта, в котором он определен.

Таким образом, тип верхнего уровня определен, по крайней мере, для всего программного текста проекта, в котором он определен.

Область действия вложенного члена определяется следующим образом:

- если член имеет открытый режим доступа (`public`), то его область действия совпадает с областью действия пространства имен или типа, в котором он определен;
- если член имеет защищенный режим доступа (`protected`), то его область действия распространяется на пересечение его области действия с областью действия типа, в которой он определен, в программном тексте этого типа и всех производных от него типов;

- если член имеет локальный режим доступа (`internal`), то его область действия совпадает с областью действия пространства имен или типа, в котором он определен, в программном тексте проекта;
- если член имеет локальный защищенный режим доступа (`internal protected`), то его область действия совпадает с областью действия пространства имен или типа, в которой он определен, в программном тексте проекта, и распространяется на пересечение области действия этого члена с областью действия типов, производных от класса, в которой он определен, в программном тексте данных типов;
- если член имеет закрытый режим доступа (`private`), то его область действия распространяется на программный текст типа, в котором он определен.

Таким образом, область действия вложенного члена, по крайней мере, включает в себя программный текст типа, в котором он определен, но не может превышать области действия данного типа.

При доступе к члену предпринимаются следующие действия:

- если член типа является недоступным, вызывается сообщение об ошибке (это не относится к членам пространств имен);
- если член имеет модификатор режима доступа `public`, доступ к нему разрешается;
- если член имеет модификатор режима доступа `protected internal`, то доступ к нему разрешается в том случае, если он производится из проекта либо из класса, в котором он определен, или из производного от него класса;
- если член имеет модификатор режима доступа `protected`, то доступ к нему разрешается в том случае, если он производится из класса, в котором он определен, или из производного от него типа;
- если член имеет модификатор режима доступа `internal`, то доступ к нему разрешается в том случае, если он производится из проекта, в котором он определен;
- если член имеет модификатор режима доступа `private`, то доступ к нему разрешается в том случае, если он производится из типа, в котором он определен;
- если не может быть применено ни одно из приведенных выше правил, выдается сообщение об ошибке.

Рассмотрим это на примере:

```
public class First_Class
{
        public          static int i;
        internal        static int j;
```

```
        private       static int k;
}
internal class Second_Class
{
        public        static int i;
        internal      static int j;
        private       static int k;
        public class Third_Class
        {
                public        static int i;
                internal      static int j;
                private       static int k;
        }
        private class Fourth_Class
        {
                public        static int i;
                internal      static int j;
                private       static int k;
        }
}
```

В этом примере:

- класс `First_Class` и статическое поле `First_Class.i` доступны в любом контексте;
- классы `Second_Class` и `Second_Class.Third_Class` и статические поля `First_Class.j`, `Second_Class.i`, `Second_Class.j`, `Second_Class.Third_Class.i` и `Second_Class.Third_Class.j` доступны только в текущем проекте;
- статическое поле `First_Class.k` доступно только в программном тексте класса `First_Class`;
- класс `Second_Class.Fourth_Class` и статическое поле `Second_Class.k` доступны программном тексте класса `Second_Class`, включающем в себя программный текст классов `Second_Class.Third_Class` и `Second_Class.Fourth_Class`;
- статическое поле `Second_Class.Third_Class.k` доступно только в программном тексте класса `Second_Class.Third_Class`;
- статические поля `Second_Class.Fourth_Class.i`, `Second_Class.Fourth_Class.j` и `Second_Class.Fourth_Class.k` доступны только в программном тексте класса `Second_Class.Fourth_Class`.

Как уже говорилось выше, при наследовании в тип включаются все члены базового класса, кроме его конструкторов и деструкторов. Это относится и к закрытым членам базового класса. Однако эти члены становятся недоступными в производном классе, и при попытке доступа к ним в этом классе будет выдано сообщение об ошибке.

Если доступ к члену, имеющему модификатор доступа `protected`, осуществляется вне программного текста класса, в котором он определен, или доступ к члену, имеющему модификатор доступа `protected internal`, осуществляется вне программного текста приложения, в котором он определен, то доступ должен осуществляться только с использованием того производного класса, в программном тексте которого осуществляется вызов. То есть в теле производного типа доступ к защищенному члену его базового класса может производиться следующим образом:

- с использованием простого имени;
- с использованием имени производного типа для образования полного имени;
- с использованием первичного выражения, имеющего производный тип, определяющего объект, содержащий данный член;
- с использованием ключевого слова `base`.

Кроме того, в инициализаторе конструктора производного типа может быть осуществлен доступ к защищенному конструктору базового класса.

Рассмотрим следующий пример:

```
public class Base_Class
{
        protected int i;
        static void Some_Func(Base_Class b, Derived_Class d)
        {
                b.x = 5;        // OK
                d.x = 6;        // OK
        }
}
public class Derived_Class: Base_Class
{
        static void Other_Func(Base_Class b, Derived_Class d)
        {
                b.x = 7;        // Ошибка
                d.x = 8;        // OK
        }
}
```

Как следует из этого примера, в программном тексте класса `Base_Class` обращение к его защищенной переменной возможно как с использованием объекта данного класса, так и объекта производного класса. Однако в программном тексте класса `Derived_Class` обращение к защищенной переменной его базового класса возможно только с использованием объекта этого класса. Использование объекта производного класса в данном случае исключено, поскольку класс `Derived_Class` не является базовым для класса `Base_Class`.

Ограничения, налагаемые на режимы доступа

Некоторые конструкции C# устанавливают ограничения на права доступа к используемым в них типам в зависимости от прав доступа к некоторому члену или типу. Для дальнейшего изложения материала определим, что права доступа к некоторому типу считаются не строже, чем права доступа к некоторому члену (в качестве которого может выступать и тип), если область действия этого некоторого типа полностью включает в себя область действия этого члена.

Ниже приведены некоторые ограничения, налагаемые на режимы доступа:

- права доступа к базовому классу любого производного класса должны быть не строже, чем права доступа к производному классу;
- права доступа к явным базовым интерфейсам любого производного интерфейса должны быть не строже, чем права доступа к производному интерфейсу;
- права доступа к типу возвращаемого значения и к типам формальных параметров типа делегата должны быть не строже, чем права доступа к самому типу делегата;
- права доступа к типу константы должны быть не строже, чем права доступа к самой константе;
- права доступа к типу поля должны быть не строже, чем права доступа к самому полю;
- права доступа к типу возвращаемого значения и к типам формальных параметров метода должны быть не строже, чем права доступа к самому методу;
- права доступа к типу свойства должны быть не строже, чем права доступа к самому свойству;
- права доступа к типу события должны быть не строже, чем права доступа к самому событию;
- права доступа к типу или типам формальных параметров индексатора должны быть не строже, чем права доступа к самому индексатору;

- права доступа к типу возвращаемого значения и к типам формальных параметров операции должны быть не строже, чем права доступа к самой операции;
- права доступа к типу возвращаемого значения и к типам формальных параметров конструктора метода должны быть не строже, чем права доступа к самому конструктору.

Рассмотрим это на примере:

```
class Base_Class
{
        ...
}
public class Derived_Class: Base_Class // Ошибка
{
        ...
}
public class Some_Class
{
                        Base_Class First_Func()             {...}
        internal        Base_Class Second_Func()     {...}
        public          Base_Class Third_Func()             {...} // Ошибка
}
```

Ошибки в рассмотренном примере связаны с тем, что для класса `Base_Class` по умолчанию использован модификатор режима доступа `internal`, определяющий более узкую область действия, чем модификатор режима доступа `public`.

Сигнатура и перегрузка членов

Методы, конструкторы, индексаторы и операции характеризуются своей сигнатурой. Сигнатура каждого из этих членов определяется по-разному:

- сигнатура метода включает в себя его имя, а также число, модификаторы и типы его формальных параметров; сигнатура метода не включает в себя тип его возвращаемого значения и модификатор `params`;
- сигнатура конструктора включает в себя число, модификаторы и типы его формальных параметров; сигнатура конструктора не включает в себя модификатор `params`;
- сигнатура индексатора включает в себя число и типы его формальных параметров; сигнатура индексатора не включает в себя тип элемента;

- сигнатура операции включает в себя имя операции, а также число и типы ее операндов параметров; сигнатура операции не включает в себя тип ее результата.

Сигнатуры членов используются при поиске их перегруженных реализаций в классах, структурах и интерфейсах:

- перегрузка методов позволяет определить в классе, структуре или интерфейсе несколько методов, имеющих одинаковое имя, но различающихся своим списком формальных параметров;
- перегрузка конструкторов позволяет определить в классе или структуре несколько конструкторов, различающихся своим списком формальных параметров;
- перегрузка индексаторов позволяет определить в классе, структуре или интерфейсе несколько индексаторов, различающихся своим списком формальных параметров;
- перегрузка операций позволяет определить в классе, структуре или интерфейсе несколько операций, использующих один и тот же знак, но различающихся своим списком операндов.

Ниже приведен пример определения сигнатур методов:

```
abstract class Some_Class
{
    abstract void Some_Func();                    // Some_Func()
    abstract void Some_Func(int i);              // Some_Func(int)
    abstract void Some_Func(ref int i);          // Some_Func(ref int)
    abstract void Some_Func(out int i);          // Some_Func(out int)
    abstract void Some_Func(int i, int j);       // Some_Func(int, int)
    abstract int Some_Func(string s);            // Some_Func(string)
    abstract int Some_Func(int i);               // Some_Func(int)
    abstract void Some_Func(string[] a);         // Some_Func(string[])
    abstract void Some_Func(params string[] a);  // Some_Func(string[])
}
```

Обратите внимание на то, что модификаторы `ref` и `out` являются неотъемлемой часть сигнатуры, что делает корректным объявление второго, третьего и четвертого методов. В то же время, модификатор `params` не является частью сигнатуры. Поэтому одновременное объявление двух последних методов некорректно.

Область видимости

Под *областью видимости* понимается фрагмент программы, в котором можно обращаться к члену типа или пространства имен по его простому имени. Области видимости могут быть вложенными, и во внутренней области видимости имя, используемое во внешней области, может быть переопределено. При этом имя, принадлежащее внешней области видимости, скрывается во внутренней области видимости. Доступ к скрытому имени становится возможен только с использованием его полного имени.

При определении области видимости действуют следующие правила:

- областью видимости члена одного пространства имен, не вложенного в другое , является программный текст единицы компиляции;
- области видимости члена одного пространства имен, вложенного в другое, являются тела всех пространств имен, полное имя которых совпадает с полным именем внешнего пространства имен или начинается с этого полного имени;
- областью видимости имени, введенного или импортированного директивой `using`, являются объявления членов пространств имен в единице компиляции или теле пространства имен, содержащего директиву `using`(директива `using` предоставляет доступ к пространствам имен или именам типов только в тех единицах компиляции или телах пространств имен, в которых она используется, и действие этой директивы не распространяется на внешние единицы компиляции и пространства имен, что, помимо прочего, исключает транзитивность);
- областью видимости члена класса является тело класса, в котором он определен. Кроме того, область видимости члена класса распространяет тело тех производных классов, в которых этот член доступен;
- областью видимости члена структуры является тело структуры, в которой он определен;
- областью видимости члена перечислимого типа является тело перечислимого типа, в котором он определен;
- областью видимости аргумента конструктора являются инициализатор и блок данного конструктора;
- областью видимости аргумента метода является блок данного метода;
- областью видимости аргумента индексатора являются процедуры доступа данного индексатора;
- областью видимости операнда перегруженной операции является блок данной операции;
- областью видимости локальной переменной, определенной в блоке, является блок, в котором она определена(объявление локальной перемен-

ной обязательно должно предшествовать ее первому использованию в данном блоке);

- областью видимости локальной переменной, определенной в инициализаторе оператора цикла `for`, являются все элементы данного оператора;
- областью видимости метки является блок, в котором она объявлена.

В пределах его области видимости к члену пространства имен, класса, структуры или перечислимого типа можно обращаться до его объявления. Например, приведенный ниже текст абсолютно корректен:

```
class Some_Class
{
        void Some_Func()
        {
                i = 0
        }
        int i;
}
```

В области видимости локальных переменных их объявление должно предшествовать их первому использованию. Рассмотрим это на примере:

```
class Some_Class
{
        void Some_Func()
        {
                i = 0;                  // Ошибка
                int i;
                int j = k = 3;          // OK
                int m = 7, n = m;       // OK
                i = 3;                  // OK
        }
        int     i;
}
```

В приведенном случае ошибка возникает только при обнулении локальной переменной `i`, поскольку эта операция производится до ее объявления. Инициализация локальных переменных `j` и `k` производится абсолютно корректно, поскольку операция присваивания выполняется справа налево. Корректность инициализации переменных `m` и `n` связана с тем, что эта операция производится слева направо.

Такие правила установления границ области видимости локальной переменной обеспечивают ее единообразную трактовку в пределах всего блока, в

котором она определена. Если бы эти правила не действовали, то данный фрагмент программы не содержал бы ошибки, но первое присваивание производилось бы полю `i` класса `Some_Class`, а второе — локальной переменной `i` метода `Some_Func` того же класса.

Скрытие имен

Область видимости имени может охватывать большую область, чем объявление данного имени. При этом область видимости может включать в себя объявления новых элементов, включающих в себя объявления членов, имеющих то же имя, что и имя, для которого определяется область видимости. Такие объявления скрывают старое имя. Если имя не скрыто в своей области видимости, то оно видимо. Скрытие имен происходит при вложении и наследовании.

Скрытие имен при вложении происходит:

- при объявлении в пространствах имен, вложенных пространств имен или типов;
- при объявлении вложенных типов в классах и структурах;
- при объявлении формальных параметров и локальных переменных.

Такое скрытие никогда не сопровождается выдачей сообщений об ошибках и предупреждениях. Скрытие имени полностью исключает его из процедуры поиска наилучшей реализации:

```
class  Demo
{
       void Some_Func()
       {
              ...
       }

       void Some_Func(int i)
       {
              ...
       }

       class First
       {
              int Some_Func;
       }
       class Second
       {
```

```
            void Some_Func(int i)
            {
                ...
            }
        }
}
```

В приведенном выше примере в классе `First` к имени `Some_Func` можно обращаться только как к полю, имеющему тип `int`, поэтому в данном классе с использованием простого имени не может быть вызвана ни одна из реализаций одноименного метода класса `Demo`. В классе `Second` переопределена только одна реализация этого метода, поэтому метод `Some_Func` без аргументов в данном классе вызываться не может.

Скрытие имен при наследовании возможно только в классах или структурах. Оно происходит в том случае, когда в производном классе или структуре объявляется не виртуальный член, имеющий то же имя, что и член базового класса. Скрытие имени при наследовании может происходить одним из следующих способов:

- константы, поля, свойства, события или типы, определенные в классе или структуре, скрывают все одноименные члены базового класса;
- методы класса или структуры скрывают все одноименные члены базового класса, не являющиеся методами, а также все методы, имеющие ту же сигнатуру (имя метода, а также число, модификаторы и типы формальных параметров);
- индексаторы класса или структуры скрывают все индексаторы базового класса, имеющие ту же сигнатуру (число и типы формальных параметров).

Поскольку, по крайней мере один из операторов перегружаемой операции должен иметь тип класса, в котором производится перегрузка, то операции никогда не скрывают друг друга, поскольку одноименные операции базового и производного класса никогда не могут иметь одну и ту же сигнатуру.

При скрытии члена базового класса компилятор выдает предупреждение. Эта ситуация не расценивается как ошибка, поскольку она может возникнуть при добавлении в базовый класс новых членов. При этом всегда может возникнуть ситуация, когда в производном классе уже определен одноименный член. Базовый класс может находиться в стандартной библиотеке классов, а производный класс — разработан пользователем, или может быть взят из другой библиотеки. Поскольку для обеспечения преемственности новые версии библиотек классов должны быть совместимы со своими старыми версиями, переход на новую версию библиотеки классов не должен привести к неработоспособности приложения, так как в нем не используются его расширенные возможности, в том числе и новый член класса, имя которого

совпало с именем члена производного класса. Однако об этом должно быть выдано предупреждение, извещающее о том, что для последующего более полного использования возможностей расширенной библиотеки в производный класс нужно внести изменения.

Для подавления этого предупреждения используется модификатор `new`, свидетельствующий о том, что разработчик производного класса знает о возникшем конфликте, но не считает нужным вносить изменения в разработанный им класс.

Как при скрытии имени при вложении, так и в случае скрытия имени при наследовании, имя скрывается только в области действия скрывающего его имени. На это особенно следует обращать внимание в случае скрытия при наследовании. Например, если скрывающий член производного класса имеет закрытый режим доступа (`private`), то в производном от него классе соответствующий член базового класса опять становится видимым:

```
class Base_Class
{
       public static void Some_Func() {}
}
class Derived_Class: Base_Class
{
       new private static void Some_Func() {}
       static void Other_Func() { Some_Func(); }
                              // Derived_Class.Some_Func
}
class More_Derived_Class: Derived_Class
{
       new static void Other_Func() { Some_Func(); }
                              // Base_Class.Some_Func
}
```

Использование имен пространств имен и типов

В определенном контексте C# требует использования имен пространств имен и типов. Оба они представляют собой один или несколько идентификаторов, разделенных точкой. Каждое из этих имен может использоваться только в своем контексте, иначе будет выдано сообщение об ошибке. То есть, нельзя использовать имя типа вместо имени пространства имен и наоборот.

Определение того, какое имя использовано в данном конкретном случае, производится следующим образом:

- Если это имя состоит из одного идентификатора:
  - если оно использовано в теле объявления класса или структуры, то, начиная с данного объявления класса или структуры и включая все внешние объявления классов или структур (если таковые имеются), производится поиск одноименного типа. Данный идентификатор рассматривается как имя этого типа;
  - иначе, начиная с пространства имен, в котором был использован данный идентификатор (если таковой имеется), включая все внешние объявления классов или структур (если таковые имеются) и глобальное пространство имен, производятся следующие действия:
  - если рассматриваемое пространство имен содержит одноименный член, то данный идентификатор, в зависимости от этого члена, рассматривается как имя типа или пространства имен;
  - в ином случае, если данное пространство имен содержит директиву `using alias`, связывающую это имя с именем импортированного типа или пространства имен, то этот идентификатор заменяется именем типа или пространства имен, указанным в директиве `using alias`;
  - если неприменимы предыдущие правила и пространство имен, импортированное директивой `using`, содержит только один тип, имеющий имя, совпадающее с данным идентификатором, то этот идентификатор рассматривается как имя данного типа;
  - если неприменимы предыдущие правила и пространство имен, импортированное директивой `using`, содержит более одного типа, имя которого совпадает с данным идентификатором, то возникает неоднозначность и выдается соответствующее сообщение об ошибке.
  - в ином случае этот идентификатор считается неопределенным и выдается сообщение об ошибке.
- В противном случае искомое имя должно состоять из нескольких идентификаторов, разделяемых точкой. При определении того, какому пространству имен или типу соответствует данное имя, оно сначала разбивается на две части: его правая часть содержит только один идентификатор, а левая часть — все остальные идентификаторы, отделяемые точкой от правого идентификатора. После этого описываемая процедура применяется для определения того, какому пространству имен или типу принадлежит правый идентификатор (то есть для раскрытия левой части разбитого имени). Затем производятся следующие действия для определения значения правого идентификатора:

- если левая часть разбитого имени определяет пространство имен, а правый идентификатор определяет доступный член этого пространства

имен, то искомое имя рассматривается как имя данного члена и, в зависимости от него, классифицируется как пространство имен или тип;

- если левая часть разбитого имени определяет тип класса или структуры, а правый идентификатор определяет доступный вложенный тип этого класса или структуры, то искомое имя рассматривается как данный вложенный тип;
- в ином случае этот идентификатор считается определенным и выдается сообщение об ошибке.

Каждое пространство имен или тип имеют полное имя, однозначно их определяющее. Полное имя типа или пространства имен определяется следующим образом:

- если пространство имен или тип принадлежат глобальному пространству имен, то его полное имя совпадает с его идентификатором;
- в ином случае полное имя состоит из полного имени пространства имен или типа, в котором содержится определение данного пространства имен или типа, за которым, через точку, следует идентификатор данного пространства имен или типа.

Другими словами, полное имя представляет собой полный путь в иерархической структуре от глобального пространства имен до определяемого пространства имен или типа. Поскольку каждый член пространства имен или типа должен иметь уникальное имя, то и полное имя каждого пространства имен или типа уникально.

Ниже приведен пример иерархической структуры пространств имен и типов с указанием полного имени каждого из ее членов:

```
class Global_Class    // Global_Class
{
        ...
}
namespace Outer_Namespace    // Outer_Namespace
{
        class First_Class     // Outer_Namespace.First_Class
        {
           class Inner_Class  // Outer_Namespace.First_Class.Inner_Class
                {
                        ...
                }
        }
        class Second_Class    // Outer_Namespace.Second_Class
        {
          class Inner_Class   // Outer_Namespace.Second_Class.Inner_Class
```

```
            {
                    ...
            }
        }
        namespace Inner_Namespace    // Outer_Namespace.Inner_Namespace
        {
          class Inner_Class   // Outer_Namespace.Second_Class.Inner_Class
            {
                    ...
            }
        }
}
  namespace Outer_Namespace.Inner_Namespace
                          // Outer_Namespace.Inner_Namespace
{
  class First_Class   // Outer_Namespace.Inner_Namespace.First_Class
        {
            ...
        }
}
```

Как видно из данного примера, одни и те же простые имена могут использоваться только в том случае, если они имеют различные полные имена.

Глава 6

Переменные

Переменные в C# представляют собой область памяти для хранения различной информации. Каждая переменная C# имеет тип, определяющий значения, которые могут быть сохранены в данной переменной. Структура языка C# и его компилятор гарантируют, что в каждой переменной может быть сохранено только значение определенного типа. Значения переменных могут быть изменены операцией присваивания или операциями инкрементирования (++) и декрементирования (--).

Перед тем, как из переменной может быть считано ее значение, оно должно быть в нее записано. Значение может присваиваться переменной в процессе ее создания. Такие переменные называются изначально инициализированными и считается, что они всегда содержат корректное значение. Если переменной в процессе ее создания не присваивается определенное значение, то эта переменная называется изначально неинициализированной. Прежде чем использовать такую переменную, значение ей необходимо присвоить. При этом непринципиально, какой путь будет использован для доступа к области памяти, соответствующей данной переменной.

Категории переменных

В C# определены следующие категории переменных:

- статические переменные;
- нестатические переменные;
- элементы массивов;
- формальные параметры, передаваемые по значению;
- формальные параметры, передаваемые по ссылке;
- возвращаемые значения;
- локальные переменные.

Статические переменные

Статической переменной является поле, объявленное с использованием модификатора `static`. Эти переменные создаются при загрузке типа, в котором они объявлены, и уничтожаются при выгрузке данного типа. Началь-

ными для статических переменных являются значения, присваиваемые им конструкторами их типов. Статические переменные всегда считаются изначально инициализированными.

Нестатические переменные

Нестатической переменной является поле, объявленное без использования модификатора `static`. Нестатические переменные классов создаются при создании новых объектов класса и уничтожаются деструктором данных объектов, вызываемым, когда на эти объекты нет ни одной ссылки. Начальными для нестатических полей классов являются значения, присваиваемые им конструкторами их типов. Нестатические поля классов всегда считаются изначально инициализированными.

Срок жизни нестатических переменных структур полностью совпадает со сроком жизни объекта структуры, в которой они определены. Другими словами, нестатическая переменная структуры создается и уничтожается одновременно с объектом той структуры, в которой она определена. Нестатическая переменная структуры инициализируется одновременно с объектом структуры, поэтому, если переменная объекта структуры изначально инициализирована, то инициализированы и ее нестатические переменные. В противном случае все эти переменные изначально не инициализированы.

Элементы массивов

Элементы массива создаются вместе с объектом массива и уничтожаются, если на этот объект нет ни одной ссылки. Начальными значениями элементов массива являются значения, присваиваемые им конструкторами их типов. Элементы массивов всегда считаются изначально инициализированными.

Формальные параметры, передаваемые по значению

Формальными параметрами, передаваемыми по значению, являются формальные параметры, не имеющие модификаторов `ref` или `out`. Эти формальные параметры создаются при вызове функционального члена (метода, конструктора, процедуры доступа или операции), для которой определен данный формальный параметр, при этом им присваивается значение соответствующего аргумента, использованного при вызове этого функционального члена. Формальный параметр уничтожается при выходе из блока соответствующего функционального члена. Формальные параметры, передаваемые по значению, всегда считаются изначально инициализированными.

Формальные параметры, передаваемые по ссылке

Формальными параметрами, передаваемыми по ссылке, являются формальные параметры, имеющие модификатор `ref`. Для этих параметров не выделяется новая область памяти, поскольку они содержат ссылку на тот же объект, что и соответствующий им аргумент, использованный при вызове данного функционального члена. Таким образом, значение формального параметра всегда совпадает с переданным ему значением.

При работе с формальными параметрами, передаваемыми по ссылке, следует придерживаться следующих правил:

- перед тем, как переменная может быть использована в качестве аргумента, соответствующего формальному параметру, передаваемому по ссылке, она должна быть полностью инициализирована;
- в блоке функционального члена формальный параметр, передаваемый по ссылке, считается изначально инициализированным.

В нестатическом методе или нестатической процедуре доступа структуры ключевое слово `this` можно рассматривать как формальный параметр, содержащий ссылку на объект структуры, для которого вызван данный метод или процедура доступа.

Возвращаемые значения

Возвращаемыми значениями являются формальные параметры, имеющие модификатор `out`. Для этих параметров не выделяется новая область памяти, поскольку они содержат ссылку на тот же объект, что и соответствующий им аргумент, использованный при вызове данного функционального члена. Таким образом, значение формального параметра всегда совпадает с переданным ему значением.

При работе с возвращаемыми значениями следует придерживаться следующих правил:

- переменная, используемая в качестве аргумента, соответствующего возвращаемому значению, может быть не инициализирована;
- после вызова функционального члена все переменные, переданные ему в аргументах, соответствующих возвращаемым значениям, считаются инициализированными;
- в блоке функционального члена возвращаемое значение считается изначально неинициализированным;
- каждое возвращаемое значение функционального члена должно быть инициализировано в его блоке до того, как будет осуществлен выход из этого блока.

В конструкторе структуры ключевое слово `this` можно рассматривать как возвращаемое значение, содержащее ссылку на создаваемый объект структуры.

Локальные переменные

Локальные переменные могут быть объявлены в блоке, операторе цикла `for` или в операторе-переключателе. Локальные переменные создаются сразу же при передаче управления блоку, оператору цикла `for` или оператору-переключателю, содержащему объявление данной переменной. Локальная переменная уничтожается при выходе из блока, оператора цикла `for` или оператора-переключателя.

Локальная переменная автоматически не инициализируется и, поэтому, не имеет значения, устанавливаемого по умолчанию. Локальная переменная всегда считается изначально неинициализированной. Если объявление локальной переменной содержит инициализатор, то эта переменная считается инициализированной во всей своей области видимости, за исключением выражения ее инициализатора.

В области видимости локальной переменной к ней нельзя обращаться до ее объявления.

Значения по умолчанию

Значения по умолчанию определены для следующих категорий переменных:

- статические переменные;
- нестатические переменные объектов классов;
- элементы массива.

Значения, устанавливаемые по умолчанию, зависят от типа переменной и определяются следующим образом:

- для переменных, имеющих тип значения, значение, устанавливаемое по умолчанию, совпадает со значением, задаваемым неявным конструктором данного типа;
- для переменных, имеющих тип ссылки, по умолчанию устанавливается значение `null`.

Инициализация

В любом месте исполняемого кода программы переменная считается инициализированной, если компилятор, используя статический анализ потоков, может определить, что переменная была автоматически инициализирована (ей было присвоено значение по умолчанию) или она хотя бы раз появилась

до этого в правой части операции присваивания. При статическом анализе потоков применяются следующие правила:

- изначально инициализированная переменная всегда рассматривается как инициализированная;
- изначально неинициализированная переменная считается инициализированной в данной точке программы, если все возможные пути выполнения программы, ведущие к данной точке, удовлетворяют по крайней мере одно из приведенных ниже требований:
  - путь содержит простую операцию присваивания, в которой данная переменная является левым операндом;
  - путь содержит вызов функционального члена или в нем создается объект, которым данная переменная передается в качестве возвращаемого значения;
  - данная переменная является локальной , и ее объявление содержит инициализатор.

Инициализация объекта структуры проверяется как для всей структуры, так и для каждой ее нестатической переменной. При этом, кроме описанной выше проверки, для типов структур производятся следующие дополнительные проверки:

- нестатическая переменная считается инициализированной, если инициализирована содержащаяся в ней переменная типа структуры;
- переменная типа структуры считается инициализированной, если в ней инициализированы все нестатические переменные.

Изначально инициализированными считаются следующие категории переменных:

- статические переменные;
- нестатические переменные объектов классов;
- нестатические переменные изначально инициализированных переменных структур;
- элементы массивов;
- формальные параметры, передаваемые по значению;
- формальные параметры, передаваемые по ссылке.

Изначально неинициализированными считаются следующие категории переменных:

- нестатические переменные изначально неинициализированных переменных структур;
- возвращаемые значения, включая переменную `this` конструкторов структур;
- локальные переменные.

Инициализация переменной требуется в следующих случаях:

- переменная должна быть инициализирована во всех точках программы, в которых используется ее значение. Выполнение этого требования гарантирует, что при этом не будет получено некорректное значение. Считается, что значение переменной используется во всех случаях, кроме перечисленных ниже:
  - переменная используется в левой части простой операции присваивания.;
  - переменная используется как аргумент, возвращающий значение;
  - переменная имеет тип структуры и используется для доступа к членам данной структуры.
- переменная должна быть инициализирована, если она используется как аргумент, соответствующий формальному параметру, передаваемому по ссылке. Выполнение этого требования гарантирует, что в блоке вызываемого функционального члена данный формальный параметр можно рассматривать как изначально инициализированный;
- все возвращаемые значения функционального члена должны быть инициализированы до того, как управление будет передано из блока данного функционального члена, независимо от того, произойдет ли это вследствие вызова оператора `return` или вследствие достижения конца данного блока. Выполнение этого требования гарантирует, что после завершения работы функционального члена все его возвращаемые значения будут инициализированы. Это позволит компилятору рассматривать передачу переменной в аргументе, возвращающем значение, эквивалентное выполнению операции присваивания значения этой переменной;
- переменная `this` в конструкторе структуры должна быть инициализирована в каждой точке, где возможен выход из данного конструктора.

Ниже приведен фрагмент программы, демонстрирующий инициализацию переменных в блоках оператора `try`:

```
class Some_Class
{
        static void Some_Func()
        {
                int i;
                try
                {
                        // Переменная i не инициализирована
                        i = 1; // Инициализация переменной i
                }
                catch
                {
```

```
                        // Переменная i не инициализирована
                        i = 1; // Инициализация переменной i
                }
                finally
                {
                        // Переменная i не инициализирована
                        i = 1; // Инициализация переменной i
                }
                // Переменная i не инициализирована
        }
}
```

Таким образом, в каждом блоке данного оператора локальную переменную приходится инициализировать заново, и после этого она так и остается неинициализированной.

При определении состояния переменной принимаются во внимание особенности выполнения условных операторов и операций `&&`, `||` и `?:`, вычисляющих значение не всех своих операндов. Рассмотрим это на примере:

```
class Some_Class
{
        static void Some_Func( bool b, int i)
        {
                int j;
                if ( b && (j = i) >= 0)
                {
                        // Переменная j инициализирована
                }
                else
                {
                        // Переменная j не инициализирована
                }
                if ( b || (j = i) >= 0)
                {
                        // Переменная j не инициализирована
                }
                else
                {
                        // Переменная j инициализирована
                }
```

```
                // Переменная j не инициализирована
        }
}
```

Поскольку второй операнд операции `&&` вычисляется только в том случае, когда первый ее операнд имеет значение `true`, то выражение `j = i`, инициализирующее переменную `j`, обязательно будет вычислено только при передаче управления первому вложенному оператору условного оператора. При передаче управления второму вложенному оператору условного оператора инициализация переменной `j` может и не произойти, поэтому компилятор рассматривает ее в данном блоке как неинициализированную.

Поскольку при выполнении первого условного оператора переменная `j` может остаться неинициализированной, то при передаче управления второму условному оператору компилятор рассматривает ее как неинициализированную. В нем имеет значение то, что второй операнд операции `||` вычисляется только в том случае, когда первый ее операнд имеет значение `false`, поэтому выражение `j = i`, инициализирующее переменную `j`, обязательно будет вычислено только при передаче управления второму вложенному оператору данного условного оператора.

Ссылки на переменные

Ссылки на переменные представляют собой выражения, классифицируемые как переменные. Они содержат ссылку на область памяти, из которой можно извлечь или в которую можно записать новое значение. В C и C++ ссылки на переменные известны под именем `lvalue`.

Ссылки на переменные используются в следующих случаях:

- в качестве левого операнда операции присваивания (которая может представлять собой вызов процедуры доступа к свойству или индексатору);
- в качестве аргумента, соответствующего формальному параметру метода, определенному с модификаторами `ref` или `out`, или формальному параметру конструктора.

Типы переменных

В C# различают три основных типа переменных:

- тип значений;
- тип ссылок;
- тип указателей.

Типы указателей могут использоваться только в небезопасных фрагментах программы для повышения ее быстродействия.

С точки зрения C# любой тип переменной можно рассматривать как объект, хотя это не совсем так. В языке C# введен специальный тип данных `object`, а все остальные типы данных считаются его прямыми или отдаленными потомками. Для того, чтобы получить возможность работать со ссылкой на переменную, имеющую тип значения, необходимо присвоить значение этой переменной другой переменной, имеющей тип `object` (произвести упаковку значения). После этого, для получения доступа к значению этой переменной нужно произвести явное преобразование типа новой переменной к требуемому типу значения (произвести его распаковку).

При присваивании значения переменой данного типа в нее копируется значение присваиваемой переменной. Это кардинально отличает эти типы от типов ссылок, в которых копируется только ссылка на присваиваемый объект.

Типы значений

Типы значений отличаются тем, что непосредственно хранят свои значения в соответствующих переменных. Эти типы можно разбить на типы структур, перечисляемые типы и простые типы.

Переменные, имеющие тип значения, всегда содержат значение данного типа. При присваивании значения данной переменной в ней сохраняется копия присваиваемого значения. Это является одним из основных отличий переменных типа значения от переменных типа ссылки, которые сохраняют копию ссылки на объект, но не копию самого объекта.

Все типы значений являются прямыми потомками класса `object`. Для получения доступа из типа значения к членам класса `object` необходимо присвоить значение этого типа переменной типа `object` (произвести упаковку значения). В отличие от типов ссылок типы значений не могут выступать в качестве базового типа для других типов, являясь, таким образом, тупиковой ветвью типов. Тип класса `object` не является единственно возможным базовым типом для типов значений, поскольку структуры могут использоваться для реализации интерфейсов, выступающих при этом в качестве базового типа структуры.

Поскольку типы значений не работают со ссылками, в них не может храниться значение `null`. Каждый тип значения имеет неявный конструктор, используемый по умолчанию, производящий инициализацию переменных данного типа. Значения, присваиваемые этими конструкторами, приведены в табл. 6.1.

Таблица 6.1. *Инициализация типов значений*

Тип	Значение при инициализации
`bool`	`false`
`byte`	`0`
`char`	`'\0'`
`decimal`	`0.0M`
`double`	`0.0D`
`enum`	Значение нулевого элемента перечислимого типа
`float`	`0.0F`
`int`	`0`
`long`	`0L`
`sbyte`	`0`
`short`	`0`
`struct`	Все поля, имеющие типы значений, инициализируются своими неявными конструкторами, а в поля, имеющие тип ссылки, записывается значение null
`Uint`	`0`
`ulong`	`0`
`Ushort`	`0`

Примечание

При объявлении локальных переменных их автоматическая инициализация не производится.

При работе с типами значений следует помнить о том, что:

- все простые типы значений являются альтернативными именами системных типов NGWS. Например, тип `long` является альтернативным именем структуры `System.Int64`;
- значения константных выражений, использующих только операнды простых типов, вычисляются в процессе компиляции;
- для инициализации простых типов могут быть использованы литералы. Например, 'C' представляет собой литерал типа `char`, а 32 — литерал типа `int`.

Инициализация переменных типов значений

Поскольку переменные типов значений являются объектами, они имеют конструкторы, используемые по умолчанию. Эти не имеющие аргументов конструкторы обнуляют создаваемые переменные. Для различных типов результат этой операции несколько различается:

- При инициализации переменных простых типов все их биты заполняются нулевыми значениями. В результате:
  - переменные типов `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong` принимают значение `0`;
  - переменная типа `char` принимает значение `'\x0000'`;
  - переменная типа `float` принимает значение `0.0f`;
  - переменная типа `double` принимает значение `0.0d`;
  - переменная типа `decimal` принимает значение `0.0m`;
  - переменная типа `bool` принимает значение `false`.
- Перечисляемый тип `E` принимает значение `0`.
- В создаваемом объекте структуры все поля, соответствующие переменным типа значения в принимают свои значения, устанавливаемые по умолчанию, а все переменные типов ссылок имеют значение `null`.

Эти конструкторы, к сожалению, не вызываются автоматически, как это имеет место в C++. Для их вызова следует использовать операцию `new`, как это показано ниже:

```
class Demo
{
        static void Main()
        {
                int i = 0;
                int j = new int();
        }
}
```

В результате обе переменные будут обнулены. Для создания переменных простых типов и перечисляемого типа могут быть использованы только конструкторы без аргументов. Для объектов структуры могут быть определены и конструкторы с аргументами. Ниже приведен пример заголовка такой структуры:

```
struct Size
{
        int h, w;
```

```
        public Size(int h, int w)
        {
                this.h = h;
                this.w = w;
        }
}
```

Структуры

Заголовок *структуры* состоит из ключевого слова `struct`, идентификатора и тела структуры. Перед ключевым словом `struct` могут располагаться набор атрибутов и набор модификаторов структуры. После идентификатора структуры может располагаться набор интерфейсов, реализованных в данной структуре. За телом структуры, представляющим собой блок, может стоять точка с запятой.

В качестве модификаторов структуры могут выступать ключевые слова:

- `new`;
- `public`;
- `protected`;
- `internal`;
- `private`.

Каждый из этих модификаторов может появиться в заголовке структуры только один раз. Модификаторы `public`, `protected`, `private` и `internal` определяют режим доступа к структуре.

Структура может содержать конструктор, константы, поля, методы, свойства, индексаторы, операции и вложенные типы.

Структуры, как правило, используются для создания простейших объектов, содержащих координаты точки, параметры прямоугольника или комбинацию основных цветов. Поскольку структура является типом значения, а класс — типом ссылки, то массив структур состоит из самих структур, а массив классов — из ссылок на объекты классов. Поэтому массив структур занимает меньше места, и доступ к нему осуществляется быстрее. С другой стороны, при передаче в качестве аргумента метода структура передается по значению, то есть в методе создается копия структуры, а класс передается по ссылке, то есть в методе используется ссылка на объект класса. Если структура имеет большой размер, использование ее в качестве аргумента замедляет работу приложения и требует выделения дополнительной памяти.

В структуре не может быть объявлен конструктор, используемый по умолчанию, то есть конструктор, не имеющий формальных параметров. Кроме того, при инициализации структуры в ней не могут быть инициализированы поля, имеющие тип ссылки. Это связано с тем, что использование конструкторов в классах и структурах несколько различается.

Перед тем, как объект класса может быть использован в приложении, он должен быть создан при помощи операции `new`. Если объект класса не создан, обращение к нему невозможно. Иначе обстоит дело со структурами. Поскольку объект структуры является самостоятельным, и с ним никакие ссылки не связываются, он может создаваться без использования ключевого слова `new`. В этом случае все поля структуры будут инициализированы нулевыми значениями. Поэтому разработчикам настоятельно рекомендуется следить за тем, чтобы при такой инициализации обеспечивалось допустимое исходное состояние объекта структуры. Это позволит избежать впоследствии множества проблем, связанных с инициализацией структур без использования операции `new`.

Использование конструктора без параметров (используемого по умолчанию), который может установить для полей значения, отличные от нулевых, привело бы к возникновению неоднозначности исходного состояния объекта структуры, что недопустимо. Поэтому использование такого конструктора запрещено.

Рассмотрим инициализацию структуры:

```
using System;
public struct MyColor
{
        public byte Red, Green, Blue;
        public MyColor(byte r, byte g, byte b)
        {
                Red    = r;
                Green  = g;
                Blue   = b;
        }
}
class Demo
{
        public static void Main()
        {
                MyColor black = new MyColor();
                MyColor white = new MyColor( 255, 255, 255);
        }
}
```

В приведенном выше примере вызывается конструктор структуры, используемый по умолчанию, и определенный в ней пользовательский конструктор структуры. Первый конструктор обнуляет все поля структуры, что соответствует черному цвету. Второй конструктор записывает в поля значения своих аргументов, соответствующих белому цвету.

Если в приведенном ниже примере заменить ключевое слово `struct` ключевым словом `class`, компилятор выдаст сообщение об ошибке:

```
using System;
public struct MyColor
{
        public byte Red, Green, Blue;
        public MyColor(byte Red, byte Green, byte Blue)
        {
                this.Red      = Red;
                this.Green    = Green;
                this.Blue     = Blue;
        }
}
class Demo
{
        public static void Main()
        {
                MyColor       Color;
                Color.Red     = 255;
                Color.Gree    = 255;
                Color.Blue    = 255;
        }
}
```

В результате этих действий объект структуры `Color` инициализируется значениями, соответствующими белому цвету.

Простые типы значений

Простые типы значений C# рассматриваются языком как простейшие структуры, определенные в пространстве имен `System`. Эти типы имеют сокращенные имена, напоминающие или повторяющие имена фундаментальных типов переменных языков C и C++, однако эти имена следует рассматривать только как альтернативные имена соответствующих структур пространства имен `System`. Эти короткие имена и имена соответствующих им структур приведены в табл. 6.2.

Таблица 6.2. *Соответствие коротких имен и имен структур*

Короткое имя	Имя структуры
sbyte	System.Sbyte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Короткое имя типа и имя соответствующей ему структуры полностью равноправны, однако использование коротких имен предпочтительнее не только с точки зрения удобства написания, но и с точки зрения удобочитаемости программ. Поскольку каждый простой тип значения представляет собой структуру, то он может использоваться для доступа к членам данной структуры. Например, переменная типа `short` может быть использована для доступа к членам структуры `System.Int16`. При этом могут быть, например, вызваны следующие методы и константы:

```
short  s = short.MinValue;  // Константа System.Int16.MinValue
char   c = s.ToChar();              // Метод System.Int16.ToChar()
int    i = s.ToInt32g();    // Метод System.Int16.ToInt32()
```

Следует помнить, что целочисленные литералы преобразуются в значения типа `int`. Поэтому для них вызываются методы структуры `System.Int32`. Например:

```
string s = 752.ToString();  // Метод System.Int32.ToString()
```

Помимо всех операций, разрешенных для типа структуры, над простыми типами значений могут быть проведены следующие дополнительные операции:

- значения многих простых типов переменных могут задаваться литералами. Например, использованный в предыдущем примере литерал `752` задает значение типа `int`, а литерал `"abc"` задает значение типа `string`. Все структуры, не являющиеся структурами простых типов значений, не мо-

гут быть инициализированы литералами. Для их создания и инициализации могут использоваться только конструкторы;

- если все операнды выражения представляют собой константы простых типов значений, то значение этого выражения вычисляется компилятором. Подобные выражения называются константными. Если в этом выражении использованы операции, определенные в других типах структур, то значения этих выражений вычисляются в процессе выполнения приложения;
- для задания констант простых типов значений используется модификатор `const`. Этот модификатор не может использоваться совместно с другими типами структур. Для того, чтобы добиться аналогичного эффекта, при объявлении полей данных типов структур может быть использована комбинация модификаторов `static readonly`;
- преобразования из типа простого типа значения и преобразования в этот тип могут использоваться в операциях преобразования типов, определенных в других типах структур. С другой стороны, ни одна пользовательская операция преобразования типа не может использоваться в другой пользовательской операции преобразования типа.

Целочисленные типы

В C# определены следующие *целочисленные типы*: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` и `char`. Эти типы имеют следующую разрядность и диапазон возможных значений:

- тип `sbyte` представляет собой 8-разрядное целое число со знаком, значения которого расположены в диапазоне от −128 до 127;
- тип `byte` представляет собой 8-разрядное целое число без знака, значения которого расположены в диапазоне от 0 до 255;
- тип `short` представляет собой 16-разрядное целое число со знаком, значения которого расположены в диапазоне от −32768 до 32767;
- тип `ushort` представляет собой 16-разрядное целое число без знака, значения которого расположены в диапазоне от 0 до 65535;
- тип `int` представляет собой 32-разрядное целое число со знаком, значения которого расположены в диапазоне от −2147483648 до 2147483647;
- тип `uint` представляет собой 32-разрядное целое число без знака, значения которого расположены в диапазоне от 0 до 4294967295;
- тип `long` представляет собой 64-разрядное целое число со знаком, значения которого расположены в диапазоне от −9223372036854775808 до 9223372036854775807;
- тип `ulong` представляет собой 64-разрядное целое число без знака, значения которого расположены в диапазоне от 0 до 18446744073709551615;

- тип `char` представляет собой 16-разрядное целое число без знака, значения которого расположены в диапазоне от 0 до 65535. Набор возможных значений типа `char` соответствует набору символов Unicode.

Операции, производимые над значениями целочисленных типов, всегда выполняются в 32-разрядном формате со знаком или без знака, или же в 64-разрядном формате со знаком или без знака. А именно:

- При выполнении унарных операций `+` и `~` операнд преобразуется к первому типу в последовательности `int`, `uint`, `long` и `ulong`, который может представить весь диапазон его возможных значений. Операция производится над преобразованным значением, и результат операции имеет тип, к которому было произведено преобразование.
- При выполнении унарной операции `-` операнд преобразуется к первому типу в последовательности `int` и `long`, который может представить весь диапазон его возможных значений. Операция производится над преобразованным значением, и результат операции имеет тип, к которому было произведено преобразование. Из этой последовательности преобразований следует, что унарная операция `-` не может быть произведена над операндом, имеющим тип `ulong`.
- При выполнении бинарных операций `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `==`, `!=`, `>`, `<`, `>=` и `<=` оба операнда преобразуются к первому типу в последовательности `int`, `uint`, `long` и `ulong`, который может представить весь диапазон возможных значений каждого операнда. Операция производится над преобразованными значениями операндов, и результат операции имеет тип, к которому было произведено преобразование, или тип `bool` для операций отношения.
- При выполнении бинарных операций `<<` и `>>` левый операнд преобразуется к первому типу в последовательности `int`, `uint`, `long` и `ulong`, который может представить весь диапазон его возможных значений. Операция производится над преобразованным значением левого операнда, и результат операции имеет тип, к которому было произведено преобразование.

Тип `char` относится к целочисленным типам, но отличается от них в двух отношениях:

- отсутствует неявное преобразование из любого другого типа в тип `char`. Это является существенным отличием языка C# от языка C++, в котором тип `char` использовался для представления одного байта в любом контексте;
- константы, имеющие тип `char` должны задаваться как символьные литералы. Символьные константы могут быть записаны в форме целочисленных литералов с использованием явного преобразования типов. Например, выражение `(char)15` эквивалентно выражению `'\x000F'`.

Для контроля переполнения при выполнении операций над целочисленными операндами используются операции и операторы `checked` и `unchecked`. Если операция над целочисленными операндами производится в операции или операторе `checked`, то при выполнении этой операции компилятором выдается сообщение об ошибке, а при ее выполнении в процессе выполнения приложения вызывается исключение `OverflowException`. Если операция, вызвавшая переполнение, выполняется в операции или операторе `unchecked`, то никакое сообщение не посылается, а в полученном результате операции отбрасываются старшие биты, которые не могут быть сохранены в формате результата.

Типы значений с плавающей точкой

К *типам значений с плавающей точкой* в C# относятся типы `float` и `double`. Тип `float` представляет собой 32-разрядный формат числа с плавающей точкой одиночной точности, а тип `double` — 64-разрядный формат числа с плавающей точкой двойной точности. Эти форматы определены в рекомендации `IEEE 754`, определяющей для них следующие значения:

- Положительное и отрицательное нулевое значение. В большинстве случаев положительное и отрицательное нулевые значения не различаются и рассматриваются как единое нулевое значение, но в некоторых случаях знак нулевого значения учитывается.
- Положительное и отрицательное значение "*бесконечность*". Это значение получается, например, при делении ненулевой величины на нулевую. При этом учитывается знак операндов.
- *Неопределенное значение.* На английском языке это значение называется нечисловым (Not a Number) и сокращенно обозначается как `NaN`. Это значение получается в тех случаях, когда результат операции не может быть однозначно определен (например, при делении одного значения "бесконечность" на другое).
- Значения, которые могут быть представлены в виде произведения трех сомножителей. Первый из них определяет знак числа и может принимать значения +1 или -1. Второй из них содержит мантиссу числа, а третий представляет собой значение 2, возведенное в некоторую степень, называемую порядком числа. Для типа `float` мантисса представляет собой целое число в диапазоне от 0 до 2^{24}, а порядок — целое число в диапазоне от -149 до 104. Для типа `double` мантисса представляет собой целое число в диапазоне от 0 до 2^{53}, а порядок — целое число в диапазоне от –1075 до 970.

Тип `float` может представлять числа в диапазоне от $1{,}5 \cdot 10^{-45}$ до $3{,}4 \cdot 10^{38}$ с точностью порядка семи значащих цифр. Тип `double` может представлять числа в диапазоне от $5{,}0 \cdot 10^{-324}$ до $1{,}7 \cdot 10^{308}$ с точностью порядка пятнадцати цифр.

Если один из операндов бинарной операции имеет тип значения с плавающей точкой, то другой ее операнд должен иметь целочисленный тип или тип значения с плавающей точкой. При этом операция выполняется следующим образом:

- если один из операндов имеет целочисленный тип, то его значение преобразуется к типу другого операнда;
- если хотя бы один из операндов имеет тип `double`, то и другой операнд преобразуется к типу `double`. Если эта операция не является операцией сравнения, возвращающей тип `bool`, результат операции также имеет тип `double`;
- если не может быть применено ни одно из приведенных выше правил, то операция производится для аргументов типа `float`. Если эта операция не является операцией сравнения, возвращающей тип `bool`, результат операции также имеет тип `float`.

Операции с плавающей точкой, включая операции присваивания, никогда не вызывают исключений. В отдельных ситуациях эти операции возвращают нулевое значение, значение "бесконечность" и неопределенное значение. При этом соблюдаются следующие правила:

- если результат операции с плавающей точкой слишком мал, чтобы представить его в требуемом формате, результатом этой операции является нулевое значение с соответствующим знаком;
- если результат операции с плавающей точкой слишком велик, чтобы представить его в требуемом формате, результатом этой операции является значение "бесконечность" с соответствующим знаком;
- если результат операции с плавающей точкой не может быть однозначно определен, то результат этой операции имеет неопределенное значение;
- если хотя бы один операнд операции с плавающей точкой имеет неопределенное значение, то результат этой операции также имеет неопределенное значение.

Операции с плавающей точкой могут выполняться с большей точностью, чем этого требует формат ее результата. Например, архитектура некоторых компьютеров позволяет использовать расширенные или длинные значения двойной точности, имеющие диапазон и точность представления своих значений, превышающие эти параметры для типа `double`. Архитектура процессора таких компьютеров может быть настроена на работу с форматом чисел с плавающей запятой, и выполнение операций с **меньшей точностью** может потребовать **больших вычислительных затрат**. В этом случае C# выполняет операции в этом расширенном типе, а затем преобразует результат к типу возвращаемого значения операции. Обычно использование более высокой разрядности при выполнении операции не сказывается на результате, однако тогда, когда промежуточные результаты вычислений хранятся в расши-

ренном формате, возможен случай, когда первая операция выводит результат за границы диапазона значений типа возвращаемого значения выражения, а последующая возвращает значение в нужный диапазон. В такой ситуации при вычислении в типе возвращаемого значения выражение имело бы значение "бесконечность", а при вычислении в соответствии с использованием расширенного типа оно может иметь конечное значение.

Десятичный тип

Десятичный тип или тип `decimal` представляет собой 128-разрядный тип данных, используемый в денежных и финансовых расчетах. Тип `decimal` может представлять числа в диапазоне от $1{,}0 \cdot 10^{-28}$ до $7{,}9 \cdot 10^{28}$ с точностью порядка 28 значащих цифр.

Десятичный тип так же, как и тип чисел с плавающей точкой содержит знак, мантиссу и порядок. Однако в данном случае порядок представляет собой не степень числа 2, а степень числа 10. Мантисса десятичного типа имеет диапазон значений от 0 до 2^{96}, а порядок может принимать значения от –28 до 0. В отличие от чисел с плавающей запятой, в десятичном типе нулевое значение не имеет знака, и в нем не определены значения "бесконечность" и неопределенное значение.

Как следует из приведенного выше описания, значение десятичного типа представляет собой 96-битовое целое число со знаком, промасштабированное степенью числа 10. Значения данного типа, меньшие 1, имеют точность представления 28 знаков после десятичной точки. Значения, большие 1, имеют точность представления 28 знаков независимо от расположения десятичной точки.

Одним из основных достоинств десятичного типа является то, что он позволяет избежать ошибок округления при представлении десятичных дробей. Это связано с тем, что в этом типе для масштабирования используется степень числа 10, совпадающая с базой десятичных дробей, в то время как в типах `float` и `double` для масштабирования используется степень числа 2, неприводимого к базе десятичных дробей и превращающего большинство десятичных дробей в бесконечные двоичные дроби.

Типы `float` и `double` не имеют неявного преобразования к типу `decimal`, поэтому, если один из операндов любой бинарной операции имеет тип `decimal`, то второй операнд должен иметь десятичный или целочисленный тип. В последнем случае целочисленное значение преобразуется к типу `decimal`.

Поскольку значения десятичного типа часто проверяются на равенство друг другу, в этом типе использованы очень строгие правила округления, позволяющие существенно снизить вероятность ошибки при проведении подобной операции. Результат любой операции над операндами десятичного типа округляется до ближайшего возможного значения (представимого в данной

разрядной сетке). Если результат операции равноудален от своих возможных значений, то в качестве округленного значения операнда принимается то из возможных значений, которое содержит более длинную последовательность нулей в своих младших разрядах.

Если в результате операции над десятичными операндами получается результат, значение которого слишком мало для представления в данном формате, ему присваивается нулевое значение. Если результат данной операции слишком велик для представления в данном формате, вызывается исключение `OverflowException`.

Десятичный тип имеет большую точность, но меньший диапазон, чем типы значений с плавающей точкой. Поэтому при преобразовании из типа значения с плавающей запятой в десятичный тип может быть вызвано исключение, а при обратном преобразовании может быть потеряна точность. Поэтому отсутствуют неявные преобразования из десятичного типа в типы значений с плавающей точкой и обратно. Для использования значений этих типов в одном выражении необходимо использовать явное преобразование, указывающее на то, что в данном случае важнее — диапазон или точность представления.

Логический тип

Логический тип или тип `bool` используется как тип значений логических выражений. Этот тип может иметь всего два значения `true` и `false`. В отличие от C++ в C# отсутствуют стандартные преобразования других типов в тип `bool`, позволяющие использовать целочисленные выражения вместо логических выражений. Отсутствуют и обратные преобразования.

В языках программирования C и C++ нулевое значение целочисленной переменной или нулевой указатель рассматривались как логическое значение `false`, а ненулевое значение целочисленной переменной или ненулевой указатель — как логическое значение `true`. В C# для получения соответствующего эффекта необходимо явным образом сравнить значение целочисленного выражения с нулем или ссылки на объект со значением `null`.

Перечислимые типы

Перечислимый тип представляет собой тип, задаваемый полным перечислением своих значений. Значениями перечислимого типа являются именованные константы типов `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong`. Если при объявлении перечислимого типа в явном виде не указан тип его констант, считается, что константы имеют тип `int`. Все константы имеют один и тот же тип, указываемый при объявлении перечислимого типа. Он называется базовым типом перечислимого типа. Тип `char` не может использоваться в качестве базового типа. Объявления перечислимых типов могут появляться в том же контексте, в котором могут появляться объявления классов.

Перечислимые типы являются потомками класса `System.Enum`, что позволяет обращаться из любого перечислимого типа к унаследованным методам и свойствам данного класса.

Объявление перечислимого типа состоит из ключевого слова `enum`, идентификатора перечислимого типа и списка именованных констант, заключенного в фигурные скобки. Элементы списка именованных констант разделяются запятыми. Перед ключевым словом `enum` могут располагаться набор атрибутов и набор модификаторов перечислимого типа. После идентификатора перечислимого типа может находиться идентификатор базового типа, отделяемый от него двоеточием, а за фигурными скобками, содержащими список именованных констант, — точка с запятой.

В качестве модификаторов делегата могут выступать ключевые слова:

- `new`;
- `public`;
- `protected`;
- `internal`;
- `private`.

Ниже приведен пример объявления перечислимого типа для дней недели, константы которого имеют тип `byte`:

```
enum Week : byte
{
        Monday,
        Tuesday,
        Wednesday,
        Thursday,
        Friday,
        Saturday,
        Sunday
}
```

В перечислимом типе может быть объявлено любое число именованных констант. При этом предъявляется требование, чтобы диапазон значений этих констант не превышал бы диапазон допустимых значений базового типа, и чтобы все константы имели бы различные имена. Именованные константы могут иметь атрибуты и могут быть в явном виде инициализированы любыми допустимыми значениями. При этом допускается, чтобы некоторые из констант имели одинаковое значение.

Преобразуем приведенный выше пример таким образом, чтобы он соответствовал европейской нумерации дней недели, нумерация начиналась бы

с единицы и имелась бы специальная константа для обозначения последнего дня недели:

```
enum Week : byte
{
        Monday         = 2,
        Tuesday        = 3,
        Wednesday      = 4,
        Thursday       = 5,
        Friday         = 6,
        Saturday       = 7,
        Sunday         = 1,
        Sabbath        = Saturday
}
```

Выше были приведены примеры неявного и явного задания значений именованных констант перечислимого типа. При явном задании значения именованной константы все просто: с использованием неявного преобразования типов присваиваемое значение преобразуется от перечислимого типа к базовому, и этим преобразованным значением инициализируется соответствующая константа. При неявной инициализации именованных констант, то есть, когда они не имеют инициализатора, им присваиваются значения, устанавливаемые по умолчанию. При этом действуют следующие правила:

- если данная именованная константа является первой в списке именованных констант, то она инициализируется нулевым значением;
- в противном случае именованная константа инициализируется значением предыдущей константы списка, увеличенной на единицу. При этом ее значение должно находится в диапазоне допустимых значений базового типа.

Таким образом, объявление перечислимого типа для европейской нумерации дней недели может быть существенно упрощено, как это показано в приведенном ниже примере.

```
using System;
enum Week: byte
{
        Monday         = 2,
        Tuesday,
        Wednesday,
        Thursday,
        Friday,
        Saturday,
        Sunday         = 1,
```

```
        Sabbath        = Saturday
}
class Demo
{
        static void Main()
        {
                for(byte b = 1; b <= (byte) Week.Sabbath; b++)
                        Console.WriteLine(GetString( b ));
        }
        static string GetString( byte b)
        {
                switch (b)
                {
                case (byte) Week.Sabbath:
                                return "Do nothing!";
                        default:
                                return "Do what you want.";
                }
        }
}
```

При инициализации именованных констант в перечислимых типах действуют те же правила, что и при определении констант в классах: допустимо использование значений других констант, если это не приводит к циклическим определениям. Учитывая правила неявной инициализации, при объявлении перечислимых типов существует намного больше возможностей создать циклическое определение.

Правила именования и определения области действия членов перечислимого типа полностью совпадают с правилами именования и определения области действия полей класса. Областью действия членов перечислимого типа является тело объявления перечислимого типа. В своей области действия для обращения к членам класса могут использоваться их простые имена. Во всех остальных случаях для обращения к членам перечислимого типа следует использовать их полные имена, содержащие имя перечислимого типа. Все члены перечислимого типа доступны только в том случае, если доступен сам перечислимый тип.

Для преобразования перечислимого типа к целочисленному типу или для преобразования одного перечислимого типа к другому перечислимому типу должны использоваться явные преобразования типов. Значения, которые может принимать перечислимый тип, не ограничиваются значениями его че-

нов. К этому типу может быть преобразовано любое значение базового типа, которое становится при этом корректным значением перечислимого типа.

Для перечислимых типов определены следующие операции: `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `^`, `&`, `|`, `~`, `++`, `--` и `sizeof`.

Типы ссылок

К типам ссылок относятся типы:

- классов;
- интерфейсов;
- массивов;
- делегатов.

Значение типа ссылки представляет собой ссылку на экземпляр типа, называемый его объектом. Любой тип ссылки может иметь значение `null`. Это значение не содержит ссылки ни на какой объект данного типа.

Классы

К типу *классов* относятся структуры данных, содержащие члены данных (константы, поля и события), функциональные члены (методы, свойства, индексаторы, перегруженные операции, конструкторы и деструкторы) и вложенные объявления типов. Типы классов поддерживают наследование: механизм, позволяющий производным классам расширять и конкретизировать функциональные возможности их базовых классов. Более подробно классы будут рассмотрены в главе 10.

См. главу 10.

Особое внимание следует обратить на предопределенные классы C#. К ним относятся классы `object` и `string`.

Класс `object` представляет собой базовый класс для всех остальных типов C#. Это значит, что любой тип C# является прямым или косвенным потомком класса `object`, и для него могут быть вызваны методы данного класса. Ключевое слово `object` является альтернативным именем класса `System.Object`.

Класс `string` является непосредственным потомком класса `object` и используется для работы со строками символов Unicode. Объекты класса `string` могут быть представлены как строковые литералы, что существенно облегчает их инициализацию. Ключевое слово `string` является альтернативным именем класса `System.String`.

Интерфейсы

Интерфейс представляет собой абстрактный тип ссылки, реализуемый в классах приложения. Членами интерфейса могут быть методы, свойства, события и индексаторы. Интерфейс не может содержать констант, полей, операций, конструкторов, деструкторов, статических конструкторов, определений типов и любых статических членов.

Поскольку интерфейсы являются абстрактными типами, их члены нуждаются в реализации, которая может осуществляться в классах и структурах. Для указания того, что некоторый интерфейс реализуется в данном классе или структуре, его имя включается в список базовых интерфейсов. Список базовых интерфейсов класса помещается за именем его базового класса и отделяется от него запятой. Если имя базового класса не указано, запятая не ставится.

Более подробно интерфейсы будут рассмотрены в главе 11.

См. главу 11.

Массивы

Массивы представляют собой структуры данных, состоящие из нескольких однотипных элементов, доступ к которым осуществляется посредством индексов. В отличие от C, где любой доступ к элементу массива, осуществляемый операцией индексирования, может быть выполнен при помощи указателя, а массивы и указатели настолько тесно связаны, что их, как правило, рассматривают совместно, в C# эта связь полностью разорвана. В отличие от C и C++, в которых память для элементов массива выделяется в стеке, в C# эта память выделяется в пространстве кучи.

Массив характеризуется своим рангом или размерностью, определяющей число индексов, которые необходимо указать для доступа к его элементу. Массив, имеющий ранг, равный единице, называется одномерным массивом. Все массивы, имеющие ранг больше единицы, называются многомерными массивами. Элемент массива может иметь любой тип, включая тип массива.

Каждое измерение массива имеет свой размер, определяющий диапазон возможных изменений соответствующего индекса массива. Индекс массива может изменяться от нуля до величины, на единицу меньшей заданного размера. Если массив не содержит элементов, размер измерения может быть нулевым. Размеры измерения не являются составной частью типа массива, а определяют объем оперативной памяти, выделяемой ему при инициализации. Число элементов массива определяется как произведение размеров его измерений.

Абстрактным базовым типом всех типов массивов является тип `System.Array`. Для каждого массива существует неявное преобразование к типу `System.Array` и явное преобразование типа `System.Array` к любому типу массива.

Следует помнить, что тип `System.Array` не является типом массива. Он представляет собой базовый класс для всех типов массива. В процессе выполнения приложения переменная типа `System.Array` может содержать значение `null` или ссылку на объект любого типа массива.

Типы массивов

Тип массива состоит из типа, не являющегося типом массива, и одного или нескольких спецификаторов ранга, представляющих собой квадратные скобки, в которых может размещаться список разделителей измерений, представляющих собой запятые. Ранг массива на единицу больше числа разделителей измерений в спецификаторе ранга.

Если тип массива содержит несколько спецификаторов ранга, то крайний правый спецификатор ранга определяет массив, имеющий тип, указанный в определении типа массива. Стоящий слева от него спецификатор ранга определяет массив ссылок на массивы, определенные стоящим справа от него спецификатором. Таким образом, массив `double[,][][,,,]` определяет тип двумерного массива, элементами которого являются одномерные массивы указателей на трехмерный массив действительных чисел. Следует отметить, что в отличие от C в C# левые спецификаторы ранга определяют массив ссылок на тип правого спецификатора ранга, а не массив указателей на этот тип.

Инициализация объекта массива

Поскольку тип массива является типом ссылки, объект этого типа должен быть не только объявлен, но и проинициализирован. Существует два способа инициализации массива: последовательный и одновременный. В обоих случаях для создания нового объекта массива используется ключевое слово `new`.

При последовательной инициализации за ключевым словом `new` следует конечный тип элемента массива (тип, не являющийся типом массива, указанный в объявлении типа массива). За ним в квадратных скобках следует список выражений, определяющий размерность данного массива. Если раскрываемый спецификатор ранга не является крайним правым, то за квадратными скобками должны следовать оставшиеся спецификаторы ранга. После них может располагаться инициализатор массива.

Последовательная инициализация позволяет раскрыть только один спецификатор ранга. Оставшиеся спецификаторы ранга раскрываются позднее для каждого элемента созданного массива ссылок по отдельности. Такая инициализация позволяет использовать для элементов массива конструкторы, вызываемые по умолчанию.

Как уже говорилось выше, для задания размеров измерений могут использоваться выражения. Эти выражения должны иметь тип `int` или тип, имеющий неявное преобразование к типу `int`.

В качестве примера рассмотрим различные способы инициализации двумерного массива. Если массив квадратный, то есть размеры его измерений постоянны, то лучше всего сразу объявить двумерный массив:

```
double[,] d = new double[50, 25];
```

Двумерный массив может быть объявлен с использованием двух спецификаторов ранга, но его уже нельзя инициализировать следующим образом:

```
double[][] d = new double[50][25];  // Ошибка
```

Вместо этого следует использовать следующий фрагмент программы:

```
double[][] d = new double[50][];
for (int i = 0; i < 50; i++)
       d[i] = new int[25];
```

Второй способ определения массива позволяет создавать в процессе работы приложения — массивы произвольной конфигурации. Например, приведенный выше фрагмент программы может быть модифицирован следующим образом:

```
double[][] d = new double[50][];
for (int i = 0; i < 50; i++)
       d[i] = new int[i+1];
```

В этом случае будет определен треугольный массив.

Если выражение последовательной инициализации массива содержит инициализатор массива, то этот инициализатор может содержать только константные выражения, а ранг и размеры измерений инициализируемого массива должны совпадать с рангом и размерами измерений инициализатора массива. Например:

```
new short[2, 3] {{0, 1, 2}, {3, 4, 5}};
```

При создании массива в процессе выполнения приложения выполняются следующие шаги:

- в порядке слева направо вычисляются значения выражений, определяющих размерность измерений массива. Полученные результаты преобразуются к типу `int` с использованием имеющихся неявных преобразований типов. Если при вычислении значения выражения или последующего преобразования типа результата к типу `int` вызывается исключение, то остальные выражения не вычисляются, и управление передается процедуре обработки данного исключения;
- производится проверка корректности полученных значений размеров измерений. Если любой из них имеет значение меньшее нуля, вызывается исключение `IndexOutOfRangeException`, и управление передается процедуре обработки данного исключения;

- выделяется оперативная память для хранения массива указанного размера. Если область оперативной памяти требуемого размера не может быть выделена, вызывается исключение `OutOfMemoryException`, и управление передается процедуре обработки данного исключения;
- для каждого элемента массива вызывается конструктор, используемый по умолчанию;
- если задан инициализатор массива, то производится вычисление значений каждого элемента инициализатора массива, и это значение присваивается соответствующему элементу массива. Вычисление значений и присваивание производится в порядке объявления выражений в инициализаторе массива. Другими словами, инициализация массива производится в порядке возрастания индексов. Первым изменяется крайний правый индекс. После того, как индекс пройдет весь диапазон своих значений, он будет обнулен и инкрементирован индекс, расположенный слева от него. Эта операция будет продолжаться до тех пор, пока не будет заполнен весь массив. Если в процессе вычисления значения или в процессе присваивания будет вызвано исключение, то инициализация массива прекратится (и все остальные элементы массива будут инициализированы своими конструкторами, используемыми по умолчанию).

Инициализаторы массива могут использоваться в объявлениях полей, локальных переменных и при создании объектов массивов. Инициализатор массива помещается в фигурные скобки, содержащие список инициализаторов элементов одного измерения массива. Во внешних фигурных скобках располагается список инициализаторов измерения массива, которому соответствует крайний левый индекс элемента массива. Если массив является многомерным, то элементами этого списка инициализаторов измерения будут списки инициализаторов измерения, соответствующего второму слева индексу элемента массива. И так далее, пока не будут исчерпаны все измерения массива. Список инициализаторов измерения, соответствующий крайнему правому индексу, содержит выражения, определяющие значения соответствующего элемента массива. Элементы списка инициализаторов разделяются запятыми.

При инициализации полей или переменных перед инициализатором может быть опущен тип инициализируемого массива, поскольку он полностью определяется типом поля или переменной. Так, например, инициализатор

```
byte[] b = {1, 2, 3, 4};
```

является сокращенной формой записи инициализатора

```
byte[] b = new byte[] {1, 2, 3, 4};
```

и эквивалентен следующему фрагменту программы:

```
byte[] b = new byte[4];
b[0]   = 1;
```

```
b[1]   = 2;
b[2]   = 3;
b[3]   = 4;
```

Инициализаторы могут использоваться только для прямоугольных массивов, поэтому элементы списков инициализаторов одного уровня должны иметь одинаковый размер и структуру независимо от того, в какой список инициализаторов они входят. То есть недопустима приведенная ниже инициализация:

```
byte[,] b = {{1},{2,3,4}};   // Ошибка
```

Размер инициализируемого массива определяется размерами используемых списков инициализаторов. Например, строка

```
byte[,] b = {{1,2},{3,4}};
```

эквивалентна следующему фрагменту программы:

```
byte[,] b = new byte[2,2];
b[0,0] = 1;
b[0,1] = 2;
b[1,0] = 3;
b[1,1] = 4;
```

При инициализации массивов, элементы которого имеют тип ссылки, необходимо помнить о том, что его элементы инициализируются величиной `null`, а не значением, определяемым типом массива. Поэтому каждый элемент такого массива нуждается в дополнительной инициализации прежде, чем он может быть использован.

Доступ к элементам массива

Для получения доступа к элементу массива необходимо указать идентификатор объекта массива, за которым в квадратных скобках должен располагаться список индексов, однозначно определяющих данный элемент массива. Для этого число элементов списка индексов должно соответствовать размерности массива, а значения всех индексов должны быть неотрицательными и быть меньше размера соответствующего измерения. В качестве элемента списка индексов может выступать любое выражение, значение которого имеет тип, имеющий неявное преобразование к типу `int`. Полученный элемент массива можно рассматривать как переменную соответствующего типа.

Для доступа к элементам массива может быть использован оператор `foreach`.

Если для любых типов ссылок `First_Type` и `Second_Type` существует явное или неявное преобразование, то это же преобразование существует для типов массивов `First_Type[Rank_Specifier]` и `Second_Type[Rank_Specifier]`, где `Rank_Specifier` — некоторый спецификатор ранга, одинаковый для обоих

массивов. Это, помимо прочего, означает, что при наличии неявного преобразования из типа `Second_Type` в тип `First_Type` переменная типа `First_Type[Rank_Specifier]` может содержать ссылку на переменную типа `Second_Type[Rank_Specifier]`. При этом автоматически проверяется допустимость подобного преобразования. Например:

```
class Demo
{
        static void Assign(object[] array, int index, object value)
        {
                array[index] = value;
        }
        static void Main()
        {
                string[] a = new string[8];
                Assign( a, 0, "Trash");
                Assign( a, 1, null);
                Assign( a, 2, 0);     // Ошибка
        }
}
```

Хотя аргументы метода `Assign` имеют абстрактные типы `object[]` и `object`, при передаче этих аргументов методу производится проверка того, что объект, на который ссылается переменная `value`, имеет тип элемента массива `array` или имеет значение `null`. При третьем вызове метода `Assign` в методе `Main` будет вызвано исключение `ArrayTypeMismatchException`, поскольку неявное преобразование типа `int` в тип `string` отсутствует.

Преобразования типов массивов не распространяются на массивы, элементы которых имеют типы значения. Например, отсутствует преобразование типа `byte[]` в тип `object[]`.

Делегаты

Делегат представляет собой структуру данных, ссылающуюся на статический метод или на объект и вызываемый для него нестатический метод. Все типы делегатов являются потомками класса `System.Delegate`.

Делегат представляет собой новый тип данных, определенный в C#. Наиболее близким аналогом к нему в языках программирования C и C++ можно считать указатель на функцию, однако, здесь имеется существенное различие: если указатель на функцию может быть использован только для статической функции, то делегат может быть создан как для статического, так и для нестатического методов.

Делегаты можно рассматривать как некую очень упрощенную форму интерфейса, используемого для задания формата одной единственной функции. Делегаты в отличие от интерфейсов, создаются не на стадии компиляции компонента, а на стадии его исполнения, что существенно повышает гибкость их использования. Спецификация делегата определяет формат метода. Для создания экземпляра делегата используется метод, имеющий заданный делегатом формат. В один делегат может быть добавлено несколько функций. В этом случае при вызове делегата вызываются все добавленные в него функции.

Объект делегата хранит всю необходимую информацию для вызова метода. Если для вызова нестатических методов достаточно только информации о самом методе, то для вызова нестатического метода необходимо хранить информацию и об объекте, для которого он вызывается. Делегат не знает и не хочет ничего знать о классе, которому принадлежит вызываемый в нем метод. Единственное, что необходимо обеспечить при вызове метода посредством делегата, это соответствие сигнатур делегата и вызываемого метода.

Объявление делегата состоит из ключевого слова `delegate`, типа возвращаемого значения, идентификатора делегата и круглых скобок, в которых может размещаться список формальных параметров. Перед ключевым словом `delegate` могут располагаться набор атрибутов и набор модификаторов делегата. После объявления делегата ставится точка с запятой.

В качестве модификаторов делегата могут выступать ключевые слова:

- `new`;
- `public`;
- `protected`;
- `internal`;
- `private`.

Каждый из этих модификаторов может появляться в объявлении делегата только один раз. Модификатор `new` может использоваться только для делегата, объявляемого в другом типе. Он подавляет предупреждение, выдаваемое при скрытии члена базового типа.

Модификаторы `public`, `protected`, `private` и `internal` определяют режим доступа к типу делегата. В некоторых случаях использование некоторых из перечисленных модификаторов недопустимо.

Список формальных параметров делегата определяет его сигнатуру. Сигнатура и тип возвращаемого значения делегата должны полностью соответствовать сигнатуре и типу возвращаемого значения любого метода, инкапсулируемого данным типом делегата. Тип делегата C# определяется его именем, а не структурой. Поэтому два типа делегата с различными идентификаторами, но одинаковой сигнатурой и типом возвращаемого значения, рассматриваются как два различных типа делегата.

Тип делегата по умолчанию имеет модификатор `sealed`, то есть этот тип не может выступать в качестве базового типа для любого другого типа. Кроме того, не допускается использовать класс `System.Delegate` в качестве базового класса для любого типа, не являющегося типом делегата. Следует отметить, что класс `System.Delegate` не является типом делегата.

Объект делегата в C# имеет особую процедуру инициализации. За исключением этой процедуры над классом или объектом делегата можно производить все операции, которые могут быть произведены над обычным классом или его объектом. Как правило, эти операции связаны с доступом к членам класса `System.Delegate` с использованием имени или идентификатора объекта класса делегата.

Все типы делегатов могут быть разбиты на две группы: *комбинируемые* и *некомбинируемые*. Комбинируемые типы делегатов должны отвечать следующим требованиям:

- делегат не должен возвращать значения (должен иметь тип возвращаемого значения `void`);
- ни один из формальных параметров делегата не должен быть объявлен как выходной формальный параметр (не должен иметь модификатор `out`).

При попытке объединить два объекта некомбинируемых типов делегатов, компилятор выдает сообщение об ошибке, если один из этих объектов не имеет значения `null`.

Хотя делегаты C# во многом аналогичны классам, для них используется специальный синтаксис создания объекта. Созданный подобным образом объект делегата ссылается на:

- статический метод;
- нестатический метод и объект, для которого он вызывается. При этом ссылка на объект не может иметь значение `null`;
- другой объект делегата.

После создания объекта делегата он уже не может ссылаться на другой объект или метод.

Для объединения делегатов используется перегруженная операция сложения, а для удаления делегата из объединения — операция вычитания. Объект делегата, созданный путем объединения двух и более объектов делегата, как минимум два из которых не имеют значения `null`, называются батарейным (*multicast*) объектом делегата. Для каждого объекта делегата его список вызова представляет собой упорядоченный список небатарейных делегатов, которые будут вызываться при вызове данного объекта делегата, а именно:

- для небатарейного объекта делегата список вызова состоит из самого объекта делегата;

- для батарейного объекта делегата, созданного объединением двух делегатов, список вызова формируется путем объединения списков вызова операндов операции сложения, использованной при этом объединении.

При вызове небатарейного делегата вызывается метод, на который он ссылается, и этому методу передается список аргументов, использованный при вызове делегата. Объект делегата возвращает то же значение, что и метод, на который он ссылается. Если в процессе выполнения метода, на который ссылается делегат, вызывается исключение, и это исключение не обрабатывается в вызванном методе, то его обработка продолжается в методе, в котором был произведен вызов делегата, как если бы метод, в котором было вызвано исключение, был вызван непосредственно, а не через делегата.

При вызове батарейного объекта делегата производится последовательный вызов всех объектов делегатов, содержащихся в его списке вызова. При каждом вызове используется один и тот же список аргументов. Если делегаты имеют аргументы, передаваемые по ссылке, то при вызове каждого метода будет использована ссылка на один и тот же объект. Изменения, внесенные в этот объект одним методом, будут переданы вызываемым за ним методам. Если в процессе выполнения метода, на который ссылается один из объектов делегатов в списке вызова батарейного объекта делегата, вызывается исключение, и это исключение не обрабатывается в вызванном методе, его обработка продолжается в методе, в котором был произведен вызов батарейного объекта делегата, как если бы метод, в котором было вызвано исключение, был вызван непосредственно, а не через делегата. При этом все последующие делегаты в списке вызова батарейного объекта делегата не вызываются.

Динамический характер делегатов позволяет использовать их в тех случаях, когда пользователь хочет изменить поведение класса в зависимости от внешних условий. Рассмотрим это на примере быстрой сортировки, программа которой приведена ниже:

```
using System;

class Demo
{
        // Объявление делегата

        public delegate bool CompareInt( int i, int j );

        // Методы быстрой сортировки

        void   Sort( CompareInt compare, ref int[] Buffer, int lo, int hi
)
        {
```

```
        int    i = lo;
        int           j = hi;
        int           x = Buffer[ (lo+hi) >> 1 ];

        while( i < j )
        {
               while( compare( Buffer[i], x ) )
                      i++;

               while( compare( x, Buffer[j] ) )
                      j--;

               if( i < j )
               {
                      int    y = Buffer[i];
                      Buffer[i++] = Buffer[j];
                      Buffer[j--] = y;
               }
        }

        if( lo < hi - 1 )
        {
               if ( lo < j )
                      Sort( compare, ref Buffer, lo, j );

               if ( i < hi )
                      Sort( compare, ref Buffer, i, hi );
        }
}

public void   Sort( CompareInt compare, ref int[] Buffer, int n)
{
        Sort( compare, ref Buffer, 0, n-1 );
}

// Методы сравнения целых чисел

static bool gt( int i, int j )
{
        return i > j;
}
```

```
        static bool lt( int i, int j )
        {
                return i < j;
        }

        public static void Main()
        {
                int           i;
                int[]  Buffer =      { 3, 2, 4, 1, 0 };
                Demo          X                    = new Demo();

                // Создание объектов делегатов

                CompareInt    decrease      = new CompareInt( gt );
                CompareInt    increase      = new CompareInt( lt );

                // Сортировка по возрастанию

                X.Sort( increase, ref Buffer, Buffer.Length );

                Console.WriteLine("Increase");

                for( i=0; i < Buffer.Length; i++)
                {
                        Console.WriteLine("Line {0} Value {1}", i,
                                          Buffer[i]);
                }

                // Сортировка по убыванию

                X.Sort( decrease, ref Buffer, Buffer.Length );

                Console.WriteLine("Decrease");

                for( i=0; i < Buffer.Length; i++)
                {
                        Console.WriteLine("Line {0} Value {1}", i,
                                          Buffer[i]);
                }
        }
}
```

В классе `Demo` объявлен делегат `CompareInt`, получающий в качестве аргументов два целых числа и возвращающий логическое значение, определяющее отношение их значений. В этом же классе объявляются две статические функции, соответствующие этому делегату. Объявленный делегат используется для сравнения элементов массива при их сортировке. Добавление в делегата той или иной функции сравнения позволяет с помощью одной и той же функции произвести сначала сортировку по возрастанию, а затем — по убыванию.

Недостатком приведенной выше программы является необходимость создания объекта делегата при вызове использующей его функции. Для того чтобы пользователю не нужно было разбираться с описанием делегатов, создание их объектов можно сделать статическими полями класса, как это сделано в приведенном ниже примере:

```
using System;

class Demo
{
        // Объявление делегата

        public delegate bool CompareInt( int i, int j );

        // Методы быстрой сортировки

        void   Sort( CompareInt compare, ref int[] Buffer, int lo, int hi
)
        {
                int    i = lo;
                int           j = hi;
                int           x = Buffer[ (lo+hi) >> 1 ];

                while( i < j )
                {
                        while( compare( Buffer[i], x ) )
                                i++;

                        while( compare( x, Buffer[j] ) )
                                j--;

                        if( i < j )
                        {
                                int    y = Buffer[i];
                                Buffer[i++] = Buffer[j];
                                Buffer[j--] = y;
                        }
                }
```

```
            if( lo < hi - 1 )
            {
                    if ( lo < j )
                            Sort( compare, ref Buffer, lo, j );

                    if ( i < hi )
                            Sort( compare, ref Buffer, i, hi );
            }
    }

    public void   Sort( CompareInt compare, ref int[] Buffer, int n)
    {
            Sort( compare, ref Buffer, 0, n-1 );
    }

    // Методы сравнения целых чисел

    static bool gt( int i, int j )
    {
            return i > j;
    }

    static bool lt( int i, int j )
    {
            return i < j;
    }

            // Создание объектов делегатов

public static readonly    CompareInt decrease = new CompareInt( gt );
public static readonly    CompareInt increase = new CompareInt( lt );

    public static void Main()
    {
            int           i;

            int[]  Buffer =      { 3, 2, 4, 1, 0 };
            Demo          X                     = new Demo();

            // Сортировка по возрастанию

            X.Sort( increase, ref Buffer, Buffer.Length );
```

```
            Console.WriteLine("Increase");

            for( i=0; i < Buffer.Length; i++)
            {
               Console.WriteLine("Line {0} Value {1}", i, Buffer[i]);
            }

            // Сортировка по убыванию

            X.Sort( decrease, ref Buffer, Buffer.Length );

            Console.WriteLine("Decrease");

            for( i=0; i < Buffer.Length; i++)
            {
               Console.WriteLine("Line {0} Value {1}", i, Buffer[i]);
            }
        }
}
```

Если класс содержит множество делегатов, большая часть которых не будет использоваться в приложении, объекты делегатов можно объявить как статические свойства класса. Для этого достаточно заменить в приведенном выше тексте объявления соответствующих статических полей следующим текстом:

```
        // Создание объектов делегатов

        public static CompareInt     decrease
        {
                get
                {

                        return (new CompareInt( gt ));
                }
        }
        public static CompareInt     increase
        {
                get
                {
                         return(new CompareInt( lt ));
                }
        }
```

Глава 7

Преобразование типов

В C# различают явное и неявное преобразования типов. При неявном преобразовании тип, к которому производится преобразование, определяется типом переменной, которой будет присвоено преобразованное значение. Поскольку любое преобразование должно выполняться однозначно и успешно завершаться, набор допустимых неявных преобразований ограничен. В тех случаях, когда необходимо указать, к какому типу следует преобразовать тип исходного значения, используется явное преобразование типов. В этом случае тип, к которому производится преобразование, должен иметь неявное преобразование к типу переменной, которой будет присвоено преобразованное значение.

Неявное преобразование типов

Различают следующие виды неявного преобразования типов:

- тождественное преобразование;
- неявное преобразование числовых типов;
- неявное преобразование перечислимых типов;
- неявное преобразование типов ссылок;
- преобразование типов значений к типу ссылки;
- неявное преобразование типов константных выражений;
- неявные преобразования пользовательских типов.

Неявное преобразование типов может производиться в различных ситуациях, включая вызов методов, получение доступа к объектам и при присваивании. Предопределенные неявные преобразования типов всегда завершаются успешно и не вызывают исключений. Этим требованиям должны отвечать и неявные преобразования пользовательских типов.

Тождественное преобразование

Тождественным называется преобразование любого типа в себя. Этот тип преобразования введен только для обеспечения полноты рассматриваемых видов преобразований.

Неявное преобразование числовых типов

Среди числовых типов имеется множество неявных преобразований. Основным принципом наличия неявного преобразования из одного типа в другой является возможность представления в новом типе всего диапазона значений старого типа. Поэтому допустимы следующие неявные преобразования типов:

- тип `sbyte` может быть преобразован к типам `short`, `int`, `long`, `float`, `double` или `decimal`;
- тип `byte` может быть преобразован к типам `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`;
- тип `short` может быть преобразован к типам `int`, `long`, `float`, `double` или `decimal`;
- тип `ushort` может быть преобразован к типам `int`, `uint`, `long`, `ulong`, `float`, `double` или `decimal`;
- тип `int` может быть преобразован к типам `long`, `float`, `double` или `decimal`;
- тип `uint` может быть преобразован к типам `long`, `ulong`, `float`, `double` или `decimal`;
- тип `long` может быть преобразован к типам `float`, `double` или `decimal`;
- тип `ulong` может быть преобразован к типам `float`, `double` или `decimal`;
- тип `char` может быть преобразован к типам `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` или `decimal`;
- тип `float` может быть преобразован к типу `double`.

Следует отметить, что преобразование значений типов `int`, `uint` или `long` к типу `float` или типа `long` к типу `double` может привести к потере точности, но не к выходу из диапазона допустимых значений. При всех остальных преобразованиях сохраняется не только диапазон допустимых значений, но и точность представления информации.

Преобразование в тип `char` может быть произведено только с использованием явного преобразования типов.

Неявное преобразование перечислимых типов

Единственным неявным преобразованием, определенным для перечислимых типов, является преобразование целочисленного литерала `0` в любой из перечислимых типов.

Неявное преобразование типов ссылок

Для типов ссылок разрешены следующие неявные преобразования:

- любой тип ссылки может быть преобразован в тип `object`;
- любой тип класса `Derived` может быть преобразован в тип класса `Base`, если класс `Derived` является потомком класса `Base`;
- любой тип класса `Derived` может быть преобразован в тип интерфейса `Base`, если класс `Derived` реализует интерфейс `Base`;
- любой тип массива `First` может быть преобразован в тип массива `Second`, если выполняются следующие условия:
  - массивы `First` и `Second` имеют одинаковую размерность;
  - элементы массивов `First` и `Second` имеют тип ссылки;
  - существует неявное преобразование из типа элемента массива `First` в тип элемента массива `Second`.
- любой тип массива может быть преобразован к типу `System.Array`;
- любой тип делегата может быть преобразован к типу `System.Delegate`;
- юбой тип массива или делегата может быть преобразован к типу `System.ICloneable`;
- тип `null` может быть преобразован к типу ссылки.

Неявные преобразования типов ссылок возможны только в том случае, если они всегда завершаются успешно, и поэтому нет необходимости проверять их результат в процессе выполнения приложения.

Явное или неявное преобразование типа ссылки никогда не изменяет тип объекта, на который осуществляется ссылка. Другими словами, изменение типа ссылки на объект не оказывает никакого влияния на сам объект.

Упаковка значений

Как уже говорилось выше, класс `object` является базовым классом для всех типов C#. Это относится и к типам значений. Таким образом, типы значений могут быть преобразованы к типу ссылки на этот класс, что позволяет рассматривать все типы C# с единой позиции и объединить их все в единую иерархическую структуру наследования.

Неявная *упаковка значения* позволяет получить доступ из этого типа к членам класса `object` и к членам интерфейса, реализованным в данном типе значения.

Если преобразование производится для значения, то для него создается объект, в который копируется данное значение. Для лучшего понимания этого процесса представим, что каждому типу значения `Value_Type` соответствует класс `Value_Type_Box`, определенный следующим образом:

```
class Value_Type_Box
{
        Value_Type    value;
        Value_Type_Box Value_Type_Box(Value_Type value)
        {
                this.value = value;
        }
}
```

В этом случае процесс преобразования значения `val` к типу ссылки сводится к выполнению выражения `(object) new Value_Type_Box(Value_Type value)`. То есть в стек записывается новая переменная типа `object`, а в куче создается объект структуры типа `Value_Type`, при этом в переменную в стеке записывается ссылка на объект в куче. Это преобразование является условным, и динамический тип преобразованной переменной совпадает с типом значения. Это может быть проверено оператором `is`, как это продемонстрировано в следующем примере:

```
int    i      = 345;
object b      = i;
if (b is int)
{
        Console.Write("OK");
}
```

В процессе этого преобразования создается копия преобразуемого значения, что принципиально отличается от преобразования типов ссылок к типу `object`, при котором значение продолжает ссылаться на тот же самый объект, но при этом он рассматривается как объект класса `object`. Поэтому результаты операций, произведенных над упакованной переменной типа значения, не будут сохранены, а те же операции, произведенные над переменной типа ссылки, сохранятся. Совершенно аналогично изменения, внесенные в переменную типа значения после ее упаковки, не будут отражены в преобразованном объекте. Рассмотрим это на примере:

```
class  Demo
{
```

```
        struct Some_Struct
        {
                public int i;
                public Some_Struct(int i)
                {
                        this.i = i;
                }
        }
        class Some_Class
        {
                public int i;
                public Some_Class(int i)
                {
                        this.i = i;
                }
        }
        static void Main()
        {
                Some_Struct   s = new Some_Struct( 1 );
                Some_Class    c = new Some_Class( 1 );

                object o = s;
                s.i = 2;
                Console.WriteLine(((Some_Struct)o).i);

                o = c;
                c.i = 2;
                Console.WriteLine(((Some_Class)o).i);
        }
}
```

В первой строке на экран будет выведено значение 1, поскольку структура относится к типам значений, а изменения значения ее поля были произведены после создания объекта. Во второй строке будет выведено значение 2, поскольку класс относится к типам ссылки, и изменения были внесены в объект, на который ссылается переменная `o`. Аналогией такого преобразования типов можно считать передачу аргумента функции C или C++ по значению или по ссылке.

Неявное преобразование типов константных выражений

При неявном преобразовании типа константного выражения допустимы следующие преобразования:

- константное выражение, имеющее тип `int`, может быть преобразовано к типу `sbyte`, `byte`, `short`, `ushort`, `uint` или `ulong`, если значение данного константного выражения находится в диапазоне допустимых значений типа, к которому производится преобразование;
- константное выражение, имеющее тип `long`, может быть преобразовано к типу `ulong`, если данное константное выражение имеет неотрицательное значение.

Неявные преобразования пользовательских типов

Преобразование пользовательского типа состоит из необязательного стандартного неявного преобразования типов, выполнения определенной пользователем операции неявного преобразования типов и последующего неявного стандартного преобразования типов. Более подробно пользовательские преобразования типов будут рассмотрены в следующем разделе.

Явное преобразование типов

К явному преобразованию типов относятся:

- все неявные преобразования;
- явное преобразование числовых типов;
- явное преобразование перечислимых типов;
- явное преобразование типов ссылок;
- явное преобразование типов интерфейсов;
- преобразование типа ссылки к типу значения;
- явные преобразования пользовательских типов.

При явном преобразовании типов необходимо в явном виде указать тип, к которому производится преобразование. Явные преобразования могут не всегда заканчиваться успешно, и при их проведении может теряться часть информации. Поэтому результат явного преобразования может зависеть от типа, к которому производится это преобразование. Явное преобразование может включать в себя и неявное преобразование, если тип, к которому производится преобразование, не совпадает с типом, в котором будет сохранен его результат.

Явное преобразование числовых типов

Явное преобразование из одного числового типа в другой применяется в том случае, если не может быть использовано неявное преобразование типов. В процессе явного преобразования числовых типов может происходить потеря информации, и могут вызываться исключения.

Разрешены следующие явные преобразования числовых типов:

- тип `sbyte` может быть преобразован к типам `byte`, `ushort`, `uint`, `ulong` или `char`;
- тип `byte` может быть преобразован к типам `sbyte` и `char`;
- тип `short` может быть преобразован к типам `sbyte`, `byte`, `ushort`, `uint`, `ulong` или `char`;
- тип `ushort` может быть преобразован к типам `sbyte`, `byte`, `short` или `char`;
- тип `int` может быть преобразован к типам `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong` или `char`;
- тип `uint` может быть преобразован к типам `sbyte`, `byte`, `short`, `ushort`, `int` или `char`;
- тип `long` может быть преобразован к типам `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong` или `char`;
- тип `ulong` может быть преобразован к типам `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` или `char`;
- тип `char` может быть преобразован к типам `sbyte`, `byte` или `short`;
- тип `float` может быть преобразован к типам `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char` или `decimal`;
- тип `double` может быть преобразован к типам `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float` или `decimal`;
- тип `decimal` может быть преобразован к типам `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float` или `double`.

Поскольку неявные преобразования типов образуют подмножество явных преобразований, то значение любого числового типа может быть преобразовано к любому другому числовому типу с использованием явного преобразования типов.

Явное преобразование числовых типов производится следующим образом:

- процесс преобразования из одного целочисленного типа в другой целочисленный тип определяется операторами или операциями `checked` и `unchecked`:

  - во вложенном операторе оператора `checked` или в аргументе операции `checked` преобразование типов производится только в том случае, если при этом не про-

изойдет потери информации, то есть, если преобразуемое значение находится в диапазоне допустимых значений типа, к которому производится преобразование. В противном случае вызывается исключение `OverflowException`, и управление передается обработчику данного исключения;

- во вложенном операторе оператора `unchecked` или в аргументе операции `unchecked` преобразование типов производится всегда. Если преобразуемое значение выходит за диапазон допустимых значений типа, к которому производится преобразование, результат такого преобразования содержит младшие биты исходного значения, которые могут быть сохранены в формате типа, к которому производится преобразование.

- при преобразовании значений типов `float`, `double` или `decimal` к одному из целочисленных типов результат округляется в направлении нуля ближайшего целого числа, и полученное целое число рассматривается как результат преобразования. Если результат преобразования выходит за диапазон допустимых значений типа, к которому производится преобразование, вызывается исключение `OverflowException` и управление передается обработчику данного исключения;

- при преобразовании типа `double` к типу `float` исходное значение округляется до ближайшего значения типа `float`. Если исходное значение слишком мало, чтобы быть представленным ненулевым значением типа `float`, результатом преобразования является положительное или отрицательное нулевое значение в зависимости от знака исходного значения. Если исходное значение выходит за диапазон допустимых значений типа `float`, то результатом является положительное или отрицательное значение "бесконечность". Если исходное значение не определено, то не определен и результат преобразования;

- при преобразовании типов `double` и `float` к типу `decimal` исходное значение преобразуется к типу `decimal` и, при необходимости, округляется по 28-му десятичному знаку. Если исходное значение слишком мало, чтобы быть представленным значением типа `decimal`, результатом преобразования является нулевое значение. Если исходное значение не определено или превышает максимально возможное значение типа `decimal`, вызывается исключение `InvalidCastException`;

- при преобразовании типа `decimal` к типам `float` или `double` исходное значение округляется до ближайшего значения типа, к которому производится преобразование. При этом никогда не вызывается исключение, даже в том случае, когда теряется точность представления преобразуемой информации.

Явное преобразование перечислимых типов

Для перечислимых типов разрешены следующие явные преобразования:

- типы `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double` или `decimal` могут быть преобразованы к значениям любого перечислимого типа;
- любой перечислимый тип может быть преобразован к типам `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double` или `decimal`;
- любой перечислимый тип может быть преобразован к любому другому перечислимому типу.

При проведении явного преобразования типов любое участвующее в нем значение перечислимого типа рассматривается как соответствующее ему значение его базового типа, после чего производится явное или неявное преобразование числовых типов.

Явное преобразование типов ссылок

Для типов ссылок разрешены следующие явные преобразования:

- тип `object` может быть преобразован к любому типу ссылки;
- любой тип класса `Base` может быть преобразован к типу класса `Derived`, если класс `Base` является базовым классом для класса `Derived`;
- любой тип класса `Class_Type` может быть преобразован к любому типу интерфейса `Interface_Type`, если класс `Class_Type` не является терминальным (его заголовок не содержит модификатора `sealed`) и если класс `Class_Type` реализует интерфейс `Interface_Type`;
- любой тип интерфейса `Interface_Type` может быть преобразован к любому типу класса `Class_Type`, если интерфейс `Interface_Type` не является терминальным (его заголовок не содержит модификатора `sealed`) и если класс `Class_Type` реализует интерфейс `Interface_Type`;
- любой тип интерфейса `First` может быть преобразован к любому другому типу интерфейса `Second`, если интерфейс `First` является потомком интерфейса `Second`;
- любой тип массива `First` может быть преобразован к любому другому типу массива `Second`, если выполняются следующие условия:
  - оба типа массива имеют одинаковую размерность;
  - элементы обоих массивов имеют тип ссылки;
  - существует явное преобразование из типа элемента массива `First` в тип элемента массива `Second`.

- тип `System.Array` может быть преобразован к любому типу массива;
- тип `System.Delegate` может быть преобразован к любому типу делегата;
- тип `System.ICloneable` может быть преобразован к любому типу массива или делегата.

Явное преобразование типов ссылок используется в тех случаях, когда в процессе преобразования необходимо провести проверку его корректности. Для успешного завершения явного преобразования типов ссылки необходимо, чтобы преобразуемая ссылка имела значение `null`, или же истинный тип преобразуемого значения допускал бы неявное преобразование к типу результата преобразования. Если требуемое преобразование не может быть выполнено, вызывается исключение `InvalidCastException`.

Явное или неявное преобразование типа ссылки никогда не изменяет тип объекта, на который осуществляется ссылка. Другими словами, изменение типа ссылки на объект не оказывает никакого влияния на сам объект.

Распаковка значения

Обратное преобразование объекта типа `object` или типа интерфейса, реализованного в данном типе значения, к типу значения (распаковка значения) может быть произведено только в том случае, если этот объект был получен путем прямого преобразования из этого типа значения в тип `object`. Эта операция представляет собой выражение, имеющее тип значения. Распаковка значения производится следующим образом:

- проверяется, что преобразуемый объект имеет тип значения, к которому производится преобразование;
- хранящееся в нем значение присваивается переменной типа значения.

Если вернуться к описанной выше модели преобразования с использованием вспомогательного типа, то выражение `(Value_Type) box`, выполняющее обратное преобразование объекта `box` к типу значения `Value_Type` можно рассматривать как выражение `((Value_Type_Box)box).value`. Если преобразуемый объект имеет значение `null` или имеет несовместимый тип, в процессе выполнения приложения вызывается исключение `InvalidCastException`.

Ниже приведен пример неправильной распаковки:

```
using System;
public class Demo
{
        public static void Main()
        {
                short i = 1;
                //Прямое преобразование
```

```
            object o = i;
            // Некорректное обратное преобразование
            try
            {
                int i = (int) o;
                Console.WriteLine("OK");
            }
            catch (InvalidCastException e)
            {
                Console.WriteLine("Ошибка");
            }
        }
}
```

Для того чтобы исправить ошибку, следует заменить строку

```
int i = (int) o;
```

строкой

```
int i = (short) o;
```

Явные преобразования пользовательских типов

Преобразование пользовательского типа состоит из необязательного стандартного явного преобразования типов, выполнения определенной пользователем операции явного или неявного преобразования типов и, возможно, из последующего неявного стандартного преобразования типов. Более подробно пользовательские преобразования типов будут рассмотрены в следующем разделе.

Преобразования пользовательских типов

Кроме преобразования стандартных типов, в C# допускается производить преобразования типов, определенных пользователем. Для преобразования этих типов используются специальные операции, которые могут быть объявлены в классах и структурах.

На преобразования пользовательских типов в C# наложены определенные ограничения. В частности, запрещено переопределять уже объявленные явные и неявные операции преобразования. В классе или структуре разреша-

ется определить преобразование из типа `Source` к типу `Target` только в том случае, если выполняются все перечисленные ниже условия:

- типы `Source` и `Target` не идентичны друг другу;
- тип `Source` или тип `Target` совпадает с типом класса или структуры, в которой определяется операция данного преобразования;
- ни один из типов `Source` или `Target` не является типом интерфейса или типом `object`;
- тип `Source` не является базовым классом для типа `Target` и тип `Target` не является базовым классом для типа `Source`.

Процесс преобразования пользовательского типа заключается в поиске наиболее подходящего для пользовательской операции преобразования типов, способного произвести требуемое преобразование. Этот процесс разбивается на несколько шагов:

- Производится поиск классов и структур, операции которых могут быть использованы для проведения данного преобразования. Набор этих классов или структур включает в себя тип исходного значения, его базовые классы, а также тип результата преобразования и его базовые классы (поскольку структуры не имеют базовых классов, то для них рассматриваются только исходный тип и тип результата).
- В полученном наборе классов или структур производится поиск пользовательских операций преобразования типов, которые могут быть использованы для проведения данного преобразования. Для того чтобы операция преобразования могла использоваться для проведения требуемого преобразования, необходимо наличие стандартного преобразования из исходного типа к типу аргумента операции и наличие стандартного преобразования из типа возвращаемого значения операции в тип результата преобразования(стандартным преобразованием называется любое явное или неявное преобразование значений, кроме определенных пользователем).
- Из всего полученного набора операций преобразований типов выбирается наиболее подходящая из них. При этом должна быть обеспечена однозначность выбора. Как правило, это преобразование требует минимального числа дополнительных преобразований. Критерии выбора наиболее подходящего преобразования будут рассмотрены в следующих разделах.

После нахождения наиболее подходящей операции преобразования осуществляется само преобразование, которое производится в три этапа:

1. Сначала, если это необходимо, производится стандартное преобразование из типа исходного значения к типу аргумента выбранной операции преобразования типов.
2. После этого выполняется сама операция преобразования.

3. И, наконец, если это необходимо, производится стандартное преобразование из типа возвращаемого значения выбранной операции преобразования типов к типу результата преобразования.

Таким образом, при преобразовании пользовательского типа может использоваться только одна пользовательская операция преобразования типов. Все остальные операции преобразования типов могут быть только стандартными.

В следующих разделах будут подробно рассмотрены явное и неявное преобразования пользовательских типов. В этих разделах будет использована следующая терминология:

- если существует стандартное неявное преобразование из типа `First` в тип `Second` и ни один из этих типов не является типом интерфейса, говорят, что тип `First` *включен* в тип `Second`, а тип `Second` включает в себя тип `First`;
- под *всеобъемлющим* типом в наборе типов понимается тип, включающий в себя все остальные типы набора. Если ни один из типов не включает в себя всех остальных типов набора, то такой набор не содержит всеобъемлющего типа. Для любого типа набора должно существовать неявное преобразование к всеобъемлющему типу данного набора;
- под *вездесущим* типом в наборе понимается тип, включенный во все остальные типы набора. Если ни один из типов не включается во все остальные типы набора, то такой набор не содержит вездесущего типа. Для вездесущего типа набора должно существовать неявное преобразование к любому другому типу данного набора.

Неявное преобразование пользовательских типов

Неявное преобразование из пользовательского типа `First` в пользовательский тип `Second` производится следующим образом:

- Производится поиск типов, содержащих пользовательские операции преобразования типов, которые могут быть использованы в данном преобразовании. Полученный набор типов обязательно содержит типы `First` и `Second`, а также базовые классы типов `First` и `Second`, если соответствующий тип является классом:
- В полученном наборе типов производится поиск пользовательских операций неявного преобразования типов, которые могут быть использованы в данном преобразовании. Для преобразования типов могут быть использованы все операции, преобразующие из типов, включающих тип `First` в типы, включенные в тип `Second`. Если в результате проведенного поиска не удалось обнаружить ни одной операции преобразования типов, отвечающей поставленным условиям, преобразование считается неопределенным и выдается сообщение об ошибке.

- В полученном наборе операций производится поиск наиболее подходящего исходного типа преобразования:
  - если какая-нибудь операция в наборе производит преобразование из типа `First`, то в качестве наиболее подходящего исходного типа преобразования выбирается тип `First`;
  - в ином случае в качестве наиболее подходящего исходного типа преобразования выбирается всеобъемлющий тип для всех исходных типов набора операций преобразования. Если среди всех исходных типов набора операций преобразования не может быть выбран один всеобъемлющий тип, то преобразование считается неоднозначным и выдается сообщение об ошибке.
- В полученном наборе операций производится поиск наиболее подходящего типа результата преобразования:
  - если какая-нибудь операция в наборе производит преобразование к типу `Second`, то в качестве наиболее подходящего типа результата преобразования выбирается тип `Second`;
  - в ином случае в качестве наиболее подходящего типа результата преобразования выбирается вездесущий тип для всех типов результата в наборе операций преобразования. Если среди всех типов результата в наборе операций преобразования не может быть выбран один вездесущий тип, то преобразование считается неоднозначным и выдается сообщение об ошибке.
- Если в результате всех вышеперечисленных операций в наборе может быть выделена одна единственная операция преобразования из наиболее подходящего исходного типа к наиболее подходящему типу результата, то эта операция считается наиболее подходящей операцией преобразования типов. Если такая операция отсутствует или если таких операций несколько, преобразование считается неоднозначным и выдается сообщение об ошибке.
- Производится преобразование пользовательских типов с использованием следующей процедуры:
  - если тип исходного значения не совпадает с наиболее подходящим исходным типом преобразования, производится соответствующее неявное преобразование типов;
  - выполняется наиболее подходящая операция преобразования типов;
  - если тип результата преобразования не совпадает с типом возвращаемого значения наиболее подходящей пользовательской операции преобразования типов, производится соответствующее неявное преобразование типов.

Явное преобразование пользовательских типов

Явное преобразование из пользовательского типа `First` в пользовательский тип `Second` производится следующим образом:

- Производится поиск типов, содержащих пользовательские операции преобразования типов, которые могут быть использованы в данном преобразовании. Полученный набор типов обязательно содержит типы `First` и `Second`, а также базовые классы типов `First` и `Second`, если соответствующий тип является классом.
- В полученном наборе типов производится поиск пользовательских операции явного и неявного преобразования типов, которые могут быть использованы в данном преобразовании. Для преобразования типов могут быть использованы все операции, преобразующие из типов, включающих тип `First`, в типы, включенные в тип `Second`. Если в результате проведенного поиска не удалось обнаружить ни одной операции преобразования типов, отвечающей поставленным условиям, преобразование считается неопределенным и выдается сообщение об ошибке;
- В полученном наборе операций производится поиск наиболее подходящего исходного типа преобразования:
  - если какая-нибудь операция в наборе производит преобразование из типа `First`, то в качестве наиболее подходящего исходного типа преобразования выбирается тип `First`;
  - в ином случае, если любая из операций набора осуществляет преобразование из типа, включающего в себя тип `First`, в качестве наиболее подходящего исходного типа преобразования выбирается всеобъемлющий тип для всех исходных типов набора операций преобразования. Если среди всех исходных типов набора операций преобразования не может быть выбран один всеобъемлющий тип, то преобразование считается неоднозначным и выдается сообщение об ошибке;
  - в ином случае в качестве наиболее подходящего исходного типа преобразования выбирается вездесущий тип для всех исходных типов набора операций преобразования. Если среди всех исходных типов набора операций преобразования не может быть выбран один вездесущий тип, то преобразование считается неоднозначным и выдается сообщение об ошибке.
- В полученном наборе операций производится поиск наиболее подходящего типа результата преобразования:
  - если какая-нибудь операция в наборе производит преобразование к типу `Second`, то в качестве наиболее подходящего типа результата преобразования выбирается тип `Second`;
  - в ином случае в качестве наиболее подходящего типа результата преобразования выбирается вездесущий тип для всех типов результата в

наборе операций преобразования. Если среди всех типов результата в наборе операций преобразования не может быть выбран один вездесущий тип, то преобразование считается неоднозначным и выдается сообщение об ошибке;

- в ином случае в качестве наиболее подходящего типа результата преобразования выбирается всеобъемлющий тип для всех типов результата в наборе операций преобразования. Если среди всех типов результата в наборе операций преобразования не может быть выбран один всеобъемлющий тип, то преобразование считается неоднозначным и выдается сообщение об ошибке;

- Если в результате всех вышеперечисленных операций в наборе может быть выделена одна единственная операция преобразования из наиболее подходящего исходного типа к наиболее подходящему типу результата, то эта операция считается наиболее подходящей операцией преобразования типов. Если такая операция отсутствует или если таких операций несколько, преобразование считается неоднозначным и выдается сообщение об ошибке;
- Производится преобразование пользовательских типов с использованием следующей процедуры:
  - если тип исходного значения не совпадает с наиболее подходящим исходным типом преобразования, производится соответствующее явное преобразование типов;
  - выполняется наиболее подходящая операция преобразования типов;
  - если тип результата преобразования не совпадает с типом возвращаемого значения наиболее подходящей пользовательской операции преобразования типов, производится соответствующее явное преобразование типов.

Преобразования объектов

Выше были приведены общие правила преобразования ссылок, но правила, относящиеся к одному типу объектов, были разбросаны по разным темам. В данном разделе будет приведено краткое резюме по преобразованию основных типов объектов.

Преобразования классов

Преобразование объекта одного класса в объект другого класса возможно только в том случае, если один из этих классов является потомком другого класса. При этом действует правило, что ссылка на объект класса может быть неявно преобразована к ссылке на объект его базового класса. При этом следует отметить, что преобразуется только тип ссылки, а не тип са-

мого объекта. Обратное преобразование возможно только в том случае, если объект, ссылка на который преобразуется, имеет тип, к которому производится преобразование или преобразование производится к одному из его базовых типов, расположенных в иерархии между исходным типом ссылки и истинным типом объекта. Рассмотрим это на примере:

```
using System;

class Base
{
        public virtual void Info()
        {
                Console.WriteLine("Base Class");
        }
}

class Intermediate : Base
{
        public override void Info()
        {
                Console.WriteLine("Intermediate Class");
        }

        public void NewFunc()
        {
                Console.WriteLine( "New Function" );
        }
}

class Derived : Intermediate
{
        public new void Info()
        {
                Console.WriteLine("Derived Class");
        }
}

class Test
{
        public static void Main()
        {
                Derived               d = new Derived();
```

```
            Base                b = d;
            object              o = d;
            Intermediate  i = (Intermediate) b;
            Derived             bd = (Derived) b;
            Derived             on = (Derived) o;

            b.Info();
            i.NewFunc();
            on.Info();
        }
}
```

В результате выполнения этой программы на экран будет выведен следующий текст:

```
Intermediate Class
New Function
Derived Class
```

Поскольку в классе `Derived` метод `Info` скрывает вызываемый метод базового класса, то при его вызове с использованием ссылки на базовый класс вызывается последняя перегрузка виртуального метода, определенного в нем перед скрытием. Хотя ссылка на объект класса `Intermediate` получается из ссылки на объект класса `Base`, по этой ссылке возможен вызов метода `NewFunc`, отсутствующего в классе `Base`. Как видно из примера, даже инициализация ссылки ссылкой на объект класса `object` позволяет получить полноценную ссылку на объект.

Преобразования интерфейсов

Интерфейсы представляют собой достаточно специфические типы, поэтому их преобразования нужно рассмотреть подробнее. Реализацию интерфейса в классе можно рассматривать как специфический вариант наследования. Поэтому допустимо преобразование ссылки на объект класса в ссылку на реализуемый в нем интерфейс. Это преобразование относится к неявным, поскольку его успешность гарантирована еще на стадии компиляции. Поскольку преобразование типа ссылки на объект не изменяет типа самого объекта, допустимо и обратное преобразование, но оно уже относится к явным преобразованиям. Пример преобразования ссылки на объект класса к типу интерфейса приведен ниже:

```
using System;

interface IDump
{
```

```
        void   Dump( string sz );
}

class Demo : IDump
{
        public void Dump( string sz )
        {
                Console.WriteLine( sz );
        }
}

class Test
{
        public static void Main()
        {
                Demo                    d = new Demo();
                IDump                   dump = d;

                dump.Dump("All right");
                Console.ReadLine();
        }
}
```

Ссылка на интерфейс неявным образом может преобразовываться к ссылке на другой интерфейс, даже не являющийся для него базовым. Такое преобразование завершается успешно только в том случае, если объект, ссылка на который преобразуется, содержит реализацию интерфейса, к которому производится преобразование.

Преобразование структур

Для структуры, помимо ее упаковки, допустимо только преобразование к типу реализуемого в ней интерфейса. При этом экземпляр структуры упаковывается в тип ссылки, который затем преобразуется к типу ссылки на соответствующий интерфейс. Явное преобразование интерфейса в структуру недопустимо.

Глава 8

Выражения

Выражением называется последовательность операций, операндов и знаков препинания, задающих некоторое вычисление. Вычисление выражений выполняется по определенным правилам преобразования, группировки, ассоциативности и приоритета, которые зависят от используемых в выражении операций, наличия круглых скобок и типов операндов. Выражениями являются:

- Значения. Каждому значению сопоставлен тип.
- Переменная. Каждой переменной сопоставлен тип, определяемый при ее объявлении.
- Пространство имен. Подобное выражение может использоваться только в качестве первичного выражения при доступе к члену. В любом другом контексте использование пространств имен в выражениях недопустимо.
- Тип. Подобное выражение может использоваться только в качестве первичного выражения или предопределенного типа при доступе к члену. В любом другом контексте использование типов в выражениях недопустимо.
- Группа методов, представляющая собой набор перегруженных методов, полученных после просмотра методов. Группа методов может быть связана с выражением, определяющим объект. При вызове метода объекта значение выражения, определяющего данный объект, присваивается переменной `this`. Группа методов может использоваться только в выражении вызова или при создании делегируемого метода. В любом другом контексте использование групп методов в выражениях недопустимо.
- Доступ к свойству. Каждый доступ к свойству имеет связанный с ним тип, совпадающий с типом самого свойства. Доступ к свойству может быть связан с выражением, определяющим объект. При вызове процедуры доступа к свойству (`set` или `get`) объекта значение выражения, определяющего данный объект, присваивается переменной `this`.
- Доступ к событию. Каждый доступ к событию имеет связанный с ним тип, совпадающий с типом самого события. Доступ к событию может быть связан с выражением, определяющим объект. Доступ к событию может осуществляться только с левой стороны от операции `+=` или `-=`. В любом другом контексте доступ к свойству недопустим.

- Доступ к индексатору. Каждый доступ к индексатору имеет связанный с ним тип, совпадающий с типом элемента индексатора. Кроме того, при доступе к индексатору должно быть указано выражение, определяющее объект, и список аргументов. При вызове процедуры доступа к индексатору (`set` или `get`) результат вычисления выражения, определяющего данный объект, присваивается переменной `this`, а результаты вычислений значений выражений в списке аргументов используются при вызове соответствующей процедуры доступа.
- Пустое выражение. Возникает вследствие вызова метода, возвращающего значение `void`. Пустое выражение может использоваться только в операторах-выражениях.

Конечным результатом выражения не может быть пространство имен, тип, группа методов или доступ к событию. Перечисленные выше выражения могут являться только составной частью других выражений.

Если в выражении осуществляется доступ к свойству или индексатору, то для выполнения этой операции вызывается соответствующая процедура доступа (`get` или `set`). Используемая процедура доступа определяется, исходя из контекста обращения.

Результат выражения

В большинстве случаев результатом вычисления выражения является его *значение*. В этих случаях выражение не может быть пространством имен, типом, группой методов или осуществлять доступ к событию. Если же выражение осуществляет доступ к свойству, индексатору или переменной, то значение этого выражения вычисляется следующим образом:

- Значением переменной является значение, хранящееся в области памяти, на которую указывает данная переменная. Перед тем как значение переменной будет в первый раз затребовано в приложении, она должна быть инициализирована. В противном случае компилятор выдаст сообщение об ошибке.
- Значение выражения, осуществляющего доступ к свойству, получается посредством вызова процедуры доступа `get` данного свойства. Если свойство не имеет процедуры доступа `get`, выдается сообщение об ошибке. В противном случае производится вызов данной процедуры доступа, и ее результат рассматривается как значение выражения.
- Значение индексатора получается посредством вызова его процедуры доступа `get`. Если индексатор не имеет процедуры доступа `get`, выдается сообщение об ошибке. В противном случае производится вызов данной процедуры доступа, которой передается список аргументов данного индексатора, и ее результат рассматривается как значение выражения.

Операции

Операции можно рассматривать как разновидность методов, именем которых являются знаки операций, а аргументами — их *операнды*. В качестве операндов могут выступать литералы, поля, локальные переменные и выражения.

В зависимости от числа аргументов различают три типа операций:

1. *Унарные* операции осуществляются над одним операндом. Унарные операции могут использовать как префиксную нотацию, когда знак операции предшествует операнду, так и постфиксную нотацию, когда знак операции стоит за операндом. Как правило префиксные операции производятся до использования значения операнда в других выражениях, а постфиксные — после использования значения операнда.
2. *Бинарные* операции осуществляются над двумя операндами. Этот тип операций наиболее распространен. Для них может использоваться только инфиксная нотация, то есть знак операции должен располагаться между операндами. Примером бинарной операции является операция сложения.
3. *Тернарные* операции осуществляются над тремя операндами. В C# определена только одна тернарная условная операция (`?:`). Тернарная операция использует инфиксную нотацию, поэтому в ней используются два знака операции.

Некоторые операции C# могут перегружаться в пользовательских классах. Это позволяет определять операции над объектами пользовательских классов и структур. Одним из операндов перегружаемой операции обязательно должен быть объект класса или структуры, в которой перегружена данная операция.

Приоритеты и порядок выполнения операций

Все операции C# имеют определенный *приоритет* и *ассоциативность*. Под ассоциативностью понимают порядок выполнения операций, принадлежащих к одной группе приоритета. Если операнд участвует сразу в нескольких операциях, то над ним производится операция, имеющая наивысший приоритет. После этого результат операции замещает свои операнды, и следующие операции производятся уже над результатом этой операции. Приоритет операций базируется на старшинстве мультипликативных операций над аддитивными операциями, установленном в математике, однако в алгоритмическом языке этот принцип существенно расширен и дополнен.

В табл. 8.1 все операции распределены по категориям, которые расположены в порядке убывания их приоритета.

Таблица 8.1. *Категории приоритета операций*

Категория	Знак операции	Описание
Первичные	( )	Скобки
	.	Выделение элемента класса или структуры
	( )	Вызов функции
	[ ]	Выделение элемента массива
	++	Постфиксное инкрементирование
	--	Постфиксное декрементирование
	new	Создание объекта
	typeof	Определение типа
	sizeof	Определение размера
	checked	Установка проверки переполнения
	unchecked	Сброс проверки переполнения
Унарные	+	Унарный плюс
	-	Изменение знака (унарный минус)
	!	Логическое отрицание
	~	Побитовое отрицание
	++	Префиксное инкрементирование
	--	Префиксное декрементирование
	()	Явное преобразование типа
Мультипликативные	*	Умножение
	/	Деление
	%	Деление по модулю
Аддитивные	+	Сложение
	-	Вычитание
Сдвиги	<<	Сдвиг влево
	>>	Сдвиг вправо
Отношения	>	Больше
	<	Меньше
	>=	Больше или равно
	<=	Меньше или равно

Таблица 8.1 *(окончание)*

Категория	Знак операции	Описание
	is	Совместимость типов
Равенство	==	Равно
	!=	Не равно
Побитовая операция И	&	Логическое И
Побитовая операция исключающее ИЛИ	^	Побитовая операция исключающее ИЛИ
Побитовая операция ИЛИ	|	Побитовая операция ИЛИ
Логическое И	&&	Логическое И
Логическое ИЛИ	||	Логическое ИЛИ
Присваивания	=	Присваивание
	*=	Умножение с присваиванием
	/=	Деление с присваиванием
	%=	Деление по модулю с присваиванием
	+=	Сложение с присваиванием
	-=	Вычитание с присваиванием
	<<=	Сдвиг влево с присваиванием
	>>=	Сдвиг вправо с присваиванием
	&=	Побитовая операция И с присваиванием
	^=	Побитовая операция исключающее ИЛИ с присваиванием
	|=	Побитовая операция ИЛИ с присваиванием

Если операнд оказывается между двух операций, принадлежащих к одной категории, операции выполняются в следующем порядке:

- все бинарные операции, за исключением операций присваивания, выполняются слева направо, то есть сначала выполняется операция, указанная слева от данного операнда;
- операции присваивания и условная операция выполняются справа налево то есть последовательность операций `x = y = z` присваивает переменным `x` и `y` значение переменной `z`.

Для изменения последовательности выполнения операций используются скобки, принадлежащие к категории, имеющей наивысший приоритет.

Перегрузка операций

Все унарные и бинарные операции имеют свою предопределенную реализацию для основных типов переменных. В тех случаях, когда необходимо обеспечить простой и интуитивно понятный интерфейс работы с объектами пользовательского класса или структуры, в них могут быть *перегружены* стандартные операции.

Перегруженная операция представляет собой метод, одним из аргументов которого является объект класса или структуры, в которой производится перегрузка данной операции. Синтаксис вызова данного метода совпадает с синтаксисом вызова перегружаемой операции. При перегрузке операции нельзя изменять число аргументов операции, то есть при перегрузке унарной операции следует задать один аргумент, а при перегрузке бинарной операции — два. Для указания того, что метод используется для перегрузки операции, перед знаком перегружаемой операции, представляющим собой имя метода, ставится ключевое слово `operator`.

Допускается перегрузка унарных операций `+`, `-`, `!`, `~`, `++`, `--`, `true` и `false` и бинарных операций `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, `<=`.

Следует обратить внимание на то, что в C# недопустима перегрузка операции присваивания, которая не только была допустима в C++, но и в некоторых случаях была неотъемлемой частью некоторых разновидностей классов. Однако следует помнить, что перегрузка двоичной операции влечет за собой перегрузку соответствующей операции присваивания. Например, перегрузка операции `<<` влечет за собой перегрузку операции `<<=`. Так что иногда перегрузка операции присваивания может быть осуществлена опосредованно. Например, операция `<<=` может использоваться для добавления нового элемента в объект класса. В этом случае операция `<<` не имеет смысла, и ее перегрузка не повлияет на функциональные возможности класса. Таким образом, в C# нельзя перегружать только операцию `=`. Эта операция для всех типов переменных осуществляется как побитовое копирование значения в переменную.

Перегрузка операции явного преобразования типа позволяет преобразовать стандартный тип к типу пользовательского класса или структуры или преобразовать этот тип в один из стандартных типов.

Операция выделения элемента массива не перегружается. Ее функции выполняют определенные в классе или структуре индексаторы.

При вызове перегруженной операции может быть использован синтаксис операции или синтаксис метода. В последнем случае перед знаком перегружаемой операции, представляющим собой имя метода, ставится ключевое слово `operator`. Например, перегруженные операции

```
i++;
```

```
-i;
i + j
```

могут быть записаны в виде

```
operator ++(i);
operator -(i);
operator +(i, j);
```

При перегрузке операций следует помнить, что при этом стирается различие между префиксными и постфиксными операциями, поскольку единственным отличительным признаком является знак операции. Таким образом, объявление префиксной операции объявляет и идентичную ей постфиксную операцию и наоборот.

Перегруженная операция относится к тому же классу приоритета, что и перегружаемая операция, и для нее продолжают действовать все правила, связанные с порядком действия операций.

Пользователь не ограничен в выборе действий, назначаемых перегружаемой операции, однако, если он не хочет запутаться сам и запутать потенциальных пользователей разрабатываемого им класса или структуры, ему следует выбирать для перегрузки операцию, стандартное действие которой максимально соответствует назначаемому ей действию в новом контексте. Например, если разрабатываемый класс используется для работы с матрицами, то для реализации сложения объектов этого класса лучше всего использовать операцию +, а для умножения — операцию *, хотя никто не запрещает перегружать для этого другие операции.

Составление списка кандидатов на реализацию перегруженной операции

Пусть над объектом типа `Some_Type` производится некоторая операция. При поиске *реализации* этой операции производятся следующие действия:

- В заголовке типа `Some_Type` производится поиск объявлений методов, начинающихся с ключевого слова `operator`, имеющих имя, совпадающее со знаком производимой операции и имеющее набор формальных параметров, совпадающий со списком аргументов производимой операции. Если найден хотя бы один такой метод, все найденные методы помещаются в список кандидатов на реализации перегруженной операции и возвращаются приложению.
- Если не найден такой метод, а проверяемый тип является типом `object`, возвращается пустой список кандидатов.
- Если метод не найден, но проверяемый тип отличен от типа `object`, данная методика применяется к базовому типу проверяемого типа.

Вызов перегруженных унарных операций

Если унарная операция применяется к выражению, имеющему тип `Some_Type`, то при поиске ее реализации выполняются следующие действия:

- По описанной выше методике для типа `Some_Type` составляется список кандидатов на реализацию перегруженной операции.
- Если полученный список кандидатов пуст, то в него помещается предопределенная реализация данной операции.
- С использованием правил перегрузки операций в списке кандидатов производится поиск операции, чей набор формальных параметров в наилучшей степени соответствует списку аргументов производимой операции. Если применение этих правил не позволило однозначно выделить наилучшего кандидата, выдается сообщение об ошибке.

Вызов перегруженных бинарных операций

Если над выражениями `ex1`, имеющим тип `Type1`, и `ex2`, имеющим тип `Type2`, производится бинарная операция, то при поиске ее реализации производятся следующие действия:

- По описанной выше методике для типов `Type1` и `Type2` составляются списки кандидатов на реализацию перегруженной операции. Полученные списки объединяются. При объединении списков производится проверка того, чтобы результирующий список не включал бы в себя два упоминания одного и того же метода. Такая ситуация может возникнуть в том случае, если типы `Type1` и `Type2` совпадают или имеют общий базовый класс.
- Если полученный список кандидатов пуст, то в него помещается предопределенная реализация данной операции.
- С использованием правил перегрузки операций в списке кандидатов производится поиск операции, чей набор формальных параметров в наилучшей степени соответствует списку аргументов производимой операции. Если применение этих правил не позволило однозначно выделить наилучшего кандидата, выдается сообщение об ошибке.

Преобразование типов операндов

При выполнении предопределенных унарных и бинарных операций для их операндов производится неявное преобразование типов. Эти правила не имеют прямого действия, а являются результатом правил перегрузки для предопределенных операций. Правила преобразования типов операндов не распространяются на перегруженные операции. Поэтому их реализация в этом случае полностью лежит на плечах программиста, перегружающего данную операцию.

Для реализации преобразования типов операндов необходимо задать несколько версий одной и той же операции, различающихся только типом аргументов и типом возвращаемого значения (тип возвращаемого значения не может быть единственным отличительным признаком). Рассмотрим это на примере перегрузки операции сложения:

```
int     operator +(int a, int b);
uint    operator +(uint a, uint b);
long    operator +(long a, long b);
ulong   operator +(ulong a, ulong b);
float   operator +(float a, float b);
double  operator +(double a, double b);
decimal operator +(decimal a, decimal b);
```

Этот набор перегруженных операций позволяет осуществлять операции не только над операндами, имеющими одинаковый тип или операндами, имеющими тип, указанный в одной из перегруженных операций. Необходимо только, чтобы оба операнда имели неявное преобразование к типу соответствующих операндов одной из перегруженных операций. Например, для сложения переменной типа `byte` с переменной типа `short` будет использована операция `operator +(int, int)` и результат этой операции будет иметь тип `int`.

При выполнении унарных операций `+`, `-` и `~` операнды, имеющие тип `sbyte`, `byte`, `short`, `ushort` или `char` преобразуются к типу `int`. Соответственно к тому же типу преобразуется и результат операции. Помимо этого, при выполнении унарной операции — операнды, имеющие тип `uint`, преобразуются к типу `int`.

Тип операндов преобразуется в предопределенных бинарных операциях `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `==`, `!=`, `>`, `<`, `>=` и `<=`. Как уже было показано выше, тип обоих операндов преобразуется к некоторому общему типу. Для операндов, имеющих тип значения, тип, к которому они преобразуются, совпадает с типом результата операции. При преобразовании типов операндов действуют следующие правила:

- Если любой из операндов имеет тип `decimal`, то и другой операнд должен быть преобразован к типу `decimal`. Поскольку типы `float` и `double` не имеют неявного преобразования к типу `decimal`, то при использовании переменных этих типов совместно с переменной типа `decimal` в перечисленных выше бинарных операциях выдается сообщение об ошибке.
- Если неприменимы предыдущие преобразования и любой из операндов имеет тип `double`, то и другой операнд должен быть преобразован к типу `double`.

- Если неприменимы предыдущие преобразования, и любой из операндов имеет тип `float`, то и другой операнд должен быть преобразован к типу `float`.
- Если неприменимы предыдущие преобразования, и любой из операндов имеет тип `ulong`, то и другой операнд должен быть преобразован к типу `ulong`. Поскольку типы `sbyte`, `short`, `int` и `long` не имеют неявного преобразования к типу `ulong`, то при использовании переменных этих типов совместно с переменной типа `ulong` в перечисленных выше бинарных операциях выдается сообщение об ошибке.
- Если неприменимы предыдущие преобразования, и любой из операндов имеет тип `long`, то и другой операнд должен быть преобразован к типу `long`.
- Если неприменимы предыдущие преобразования, и один из операндов имеет тип `uint`, а другой операнд имеет тип `sbyte`, `short` или `int`, то оба операнда преобразуются к типу `long`.
- Если неприменимы предыдущие преобразования, и любой из операндов имеет тип `uint`, то и другой операнд должен быть преобразован к типу `uint`.
- Если неприменимы предыдущие преобразования, то оба операнда должны быть преобразованы к типу `int`.

Из описанных выше правил преобразования следует, что операнд типа `decimal` несовместим с типами переменных и констант с плавающей запятой, а тип `ulong` несовместим с типами целых чисел со знаком. В обоих случаях для устранения возникающей ошибки может быть использовано явное преобразование типов. Однако следует помнить, что используя явное преобразование типов, программист берет на себя ответственность за возникающие при этом неоднозначности. Так, например, в результате выполнения следующей программы на экран будет выведено значение -1, хотя оба сомножителя положительные:

```
class Derived_Class
{
        static void Main()
        {
                int     n = 1;
                uint    u = 0xFFFFFFFF;

                n = n*(int)u;

                Console.WriteLine("n = {0}", n);
        }
}
```

Доступ к члену типа

Доступ к члену типа в пределах своего типа осуществляется по его простому имени. Вне пределов этого типа доступ к члену данного типа осуществляется по его полному имени, состоящему из имени типа или объекта и простого имени.

Поиск члена `Member` типа `Some_Type` производится следующим образом:

- составляется список всех доступных членов класса или структуры `Some_Type`, а также членов их базовых типов, имеющих имя `Member`;
- если список пуст, то считается, что соответствующий член не найден, и процедура поиска завершается;
- из списка исключаются объявления членов, имеющих модификатор `override`;
- из списка исключаются методы базовых классов, скрытые методами производных классов, при этом для каждого элемента списка производятся следующие действия:
  - если `Member` является константой, полем, свойством, событием, типом или элементом перечислимого типа, то из списка удаляются все члены базовых классов, того класса, членом которого является `Member`;
  - если `Member` является методом, то в списке сохраняются только те методы базовых классов, того класса, членом которого является `Member`, которые имеют отличный от него список параметров.
- после удаления скрытых членов при определении члена класса или структуры, к которому осуществляется доступ, используются следующие правила:
  - если в результате проведенных выше операций в списке остался только один член, не являющийся методом, то он и считается результатом поиска;
  - если в результате проведенных выше операций в списке остались только методы, то эта группа методов и считается результатом поиска;
  - если же не могут быть применены приведенные выше правила, то компилятор выдает сообщение об ошибке (это возможно только в том случае, если поиск производится в интерфейсе, имеющем несколько базовых интерфейсов).

При поиске члена типа, не являющегося интерфейсом, или члена интерфейса, не использующего множественного наследования (все интерфейсы в его иерархии имеют не более одного базового интерфейса), применение этих правил позволяет определить тип, в котором данный член, не являющийся методом, скрывает все члены его базовых типов, или же полный на-

бор методов, имеющих одинаковое имя, но разный набор формальных параметров, скрывших все одноименные методы своих базовых классов. В этом случае не может возникнуть никакой неоднозначности.

При определении базовых типов для типа `Some_Type` действуют следующие правила:

- если `Some_Type` является типом `object`, то `Some_Type` не имеет базового типа;
- если `Some_Type` является типом значения, то его базовым типом является тип класса `object`;
- если `Some_Type` является типом класса, то базовыми типами `Some_Type` являются все его базовые классы, включая класс `object`;
- если `Some_Type` является типом интерфейса, то базовыми типами `Some_Type` являются все его базовые интерфейсы и тип класса `object`;
- если `Some_Type` является типом массива, то его базовыми типами являются типы классов `System.Array` и `object`;
- если `Some_Type` является делегируемым типом, то его базовыми типами являются типы классов `System.Delegate` и `object`.

Функциональные члены

К *функциональным членам* типа относятся все его члены, содержащие исполняемые выражения. Функциональные члены всегда являются членами типа и не могут быть членами пространства имен. В C# определены следующие категории функциональных членов:

- конструкторы;
- методы;
- свойства;
- индексаторы;
- перегруженные операции.

Выражения, содержащиеся в функциональных членах, выполняются при вызове соответствующих членов. Синтаксис вызова функционального члена зависит от его категории. Однако всех их объединяет то, что они представляют собой выражение, могут принимать аргументы, позволяют вычислять некоторое значение и возвращать его.

Набор формальных параметров функционального члена определяет, какие значения или ссылки должны быть переданы в списке аргументов при вызове данного функционального члена.

В табл. 8.2 приведены примеры вызова различных функциональных членов и дано описание действий, производимых при этом вызове. В этой таблице

`Some_Type` обозначает некоторый тип, `Some_Func` — некоторую функцию, `Some_Object` — некоторый объект типа, `Arguments` — список аргументов, `Some_Property` — некоторое свойство, `Some_Value` — некоторое значение.

Таблица 8.2. *Описание функциональных членов*

Категория	Пример	Описание
Вызов конструктора	`new Some_Type (Arguments)`	Выбирается конструктор класса или структуры Some_Type, набор формальных параметров которого наилучшим образом соответствует списку аргументов Arguments. При вызове конструктора ему передается список аргументов Arguments
Вызов метода	`Some_Func (Arguments)`	С использованием правил перегрузки выбирается метод класса или структуры, в которой произведен вызов данного метода, набор формальных параметров которого наилучшим образом соответствует списку аргументов Arguments. При вызове метода ему передается список аргументов Arguments. Если метод не является статическим, для формирования его полного имени используется ключевое слово this
	`Some_Type.Some_Func(Arguments)`	С использованием правил перегрузки выбирается метод класса или структуры Some_Type, набор формальных параметров которого наилучшим образом соответствует списку аргументов Arguments. Если этот метод не является статическим, выдается сообщение об ошибке. При вызове метода ему передается список аргументов Arguments
	`Some_Object.Some_Func(Arguments)`	С использованием правил перегрузки выбирается метод класса, структуры или интерфейса, определяемого объектом Some_Object, набор формальных параметров которого наилучшим образом соответствует списку аргументов Arguments. Если этот метод является статическим, выдается сообщение об ошибке. Метод вызывается для объекта Some_Object и списка аргументов Arguments
Доступ к свойству	`Some_Property`	Вызывается процедура доступа get для свойства Some_Property класса или структуры, в которой произведено обращение к данному свойству. Если свойство может быть использовано только для записи, выдается сообщение об ошибке. Если свойство не является статическим, для формирования его полного имени используется ключевое слово this

Таблица 8.2 *(продолжение)*

Категория	Пример	Описание
	`Some_Property = Some_Value`	Вызывается процедура доступа set для свойства Some_Property класса или структуры, в которой произведено обращение к данному свойству. В качестве аргумента ей передается значение Some_Value. Если свойство может быть использовано только для чтения, выдается сообщение об ошибке. Если свойство не является статическим, для формирования его полного имени используется ключевое слово this
	`Some_Type.Some_Property`	Вызывается процедура доступа get для свойства Some_Property класса или структуры Some_Type. Если свойство может быть использовано только для записи или не является статическим, выдается сообщение об ошибке
	`Some_Type.Some_Property = Some_Value`	Вызывается процедура доступа set для свойства Some_Property класса или структуры Some_Type. В качестве аргумента ей передается значение Some_Value. Если свойство может быть использовано только для чтения или не является статическим, выдается сообщение об ошибке
	`Some_Object.Some_Property`	Для объекта Some_Object вызывается процедура доступа get для свойства Some_Property класса, структуры или интерфейса, определяемого объектом Some_Object. Если свойство может быть использовано только для записи или является статическим, выдается сообщение об ошибке
	`Some_Object.Some_Property = Some_Value`	Для объекта Some_Object вызывается процедура доступа set для свойства Some_Property класса, структуры или интерфейса, определяемого объектом Some_Object. В качестве аргумента ей передается значение Some_Value. Если свойство может быть использовано только для чтения или является статическим, выдается сообщение об ошибке
Доступ к индексатору	`Some_Object [Arguments]`	С использованием правил перегрузки выбирается индексатор класса, структуры или интерфейса, определяемого объектом Some_Object. Для объекта Some_Object вызывается процедура доступа get выбранного индексатора, которой передается список аргументов Arguments. Если индексатор используется только для записи, выдается сообщение об ошибке

Таблица 8.2 *(окончание)*

Категория	Пример	Описание
	`Some_Object [Arguments] = Some_Value`	С использованием правил перегрузки выбирается индексатор класса, структуры или интерфейса, определяемого объектом Some_Object. Для объекта Some_Object вызывается процедура доступа set выбранного индексатора, которой передается список аргументов Arguments и значение Some_Value. Если индексатор используется только для чтения, выдается сообщение об ошибке
Вызов унарной операции	`Some_Object++`	С использованием правил перегрузки выбирается унарная операция класса или структуры, определяемой объектом Some_Object. Для объекта Some_Object вызывается выбранная операция, в качестве аргумента которой передается объект Some_Object.
Вызов бинарной операции	`Some_Object+Other_Object`	С использованием правил перегрузки выбирается бинарная операция класса или структуры, определяемой объектами Some_Object и Other_Object. Вызывается выбранная операция, в качестве аргументов которой передаются объекты Some_Object и Other_Object

Список аргументов

При вызове каждого функционального члена ему передается *список аргументов*, содержащий фактические значения переменных, определенных в наборе формальных параметров данного метода. Синтаксис задания списка аргументов при вызове функционального члена зависит от категории, к которой он принадлежит.

Список аргументов конструктора и метода может содержать выражения и переменные, имеющие тип ссылки. Элементы списка аргументов отделяются друг от друга запятой. Перед переменной, имеющей тип ссылки, в списке аргументов может стоять модификатор `ref` или `out`. Список аргументов может не содержать ни одного аргумента. Это бывает в том случае, если данный метод используется для обработки внутренних объектов типа, а тип, для которого вызывается конструктор, производит свою инициализацию независимо от предпочтений пользователя.

Выражение, используемое в списке аргументов, должно иметь тип значения. Модификатор `ref`, устанавливаемый перед переменной, имеющей тип ссылки, указывает на то, что в данном аргументе будет передана ссылка. Перед тем как переменная может быть использована в качестве аргумента,

передаваемого по ссылке, она должна быть должным образом проинициализирована. Модификатор `out`, устанавливаемый перед переменной, имеющей тип ссылки, указывает на то, что в данном аргументе будет передан результат выполнения функции. Переменная считается должным образом проинициализированной после использования ее в списке аргументов с модификатором `out`.

При вызове функционального члена в процессе выполнения приложения значения в списке аргументов вычисляются слева направо. При этом:

- Производится вычисление значений выражений и преобразование их к типу, указанному в наборе формальных параметров, с использованием неявных преобразований. Полученными значениями инициализируются соответствующие переменные набора формальных параметров, используемые в процессе работы функции.
- Для аргументов, передаваемых по ссылке, или аргументов, используемых для передачи результатов обработки, определяется область памяти, на которую ссылается данный аргумент, и эта ссылка записывается в соответствующий формальный параметр. Если в качестве подобного аргумента передается элемент массива, то в процессе выполнения приложения производится проверка того, что тип элемента массива идентичен типу формального параметра. Если эта проверка завершается неудачно, вызывается исключение `ArrayTypeMismatchException`.

Порядок вычисления значений аргументов функциональных членов определяет, что программа

```
class Demo
{
        void Some_Func(int x, int y, int z)
        {
                Console.WriteLine("i = {0}, j = {1}, k = {2}", i, j, k);
        }
        static void Main()
        {
                int i = 0;
                Some_Func(i++, i++, i++);
        }
}
```

выведет на экран строку

```
i = 0, j = 1, k = 2
```

Производимая в процессе выполнения приложения проверка типов элементов массива необходима потому, что в C# допускается, чтобы в переменной,

имеющей тип массива `Some_Type[]`, хранилась бы ссылка на объект массива `Other_Type[]`. Для этого требуется только, чтобы имелось неявное преобразование типа ссылки `Other_Type` в тип ссылки `Some_Type`. Однако при вызове функционального члена требуется, чтобы тип передаваемого аргумента был бы идентичен типу формального параметра. Рассмотрим это на примере:

```
class Demo
{
        static void Some_Func(ref object obj )
        {
.               // Обработка
        }
        static void Main()
        {
                object[] x = new object[16];
                object[] y = new int[32][];

                Some_Func(ref x[0]);      // Все в порядке
                Some_Func(ref y[1]);      // Вызывается исключение Ar-
rayTypeMismatchException
        }
}
```

Тип массива является типом ссылки и поэтому имеет неявное преобразование к типу `object`, однако, это преобразование не может быть использовано при передаче его в качестве аргумента метода. Поэтому во втором случае будет вызвано исключение `ArrayTypeMismatchException`.

Поиск реализации

При вызове функциональных членов типов производится поиск их реализации. Из всех возможных доступных реализаций выбирается та, набор формальных параметров которой максимально соответствует списку аргументов вызываемого функционального члена.

Процедура составления списка возможных реализаций отличается для каждой разновидности функциональных членов. Однако после составления этого списка к нему применяется одна и та же процедура выбора наилучшего кандидата:

- Сначала из полученного списка исключаются все кандидаты, не удовлетворяющие следующим требованиям:
  - число формальных параметров кандидата равно числу аргументов вызываемого функционального члена;

- модификаторы формальных параметров кандидата совпадают с модификаторами аргументов вызываемого функционального члена;
- для формальных параметров, имеющих тип значения, существуют неявные преобразования типов соответствующих аргументов вызываемого метода;
- для формальных параметров, имеющих модификаторы `ref` или `out` тип аргумента идентичен типу формального параметра.

Если после этой операции список кандидатов не содержит ни одного элемента, выдается сообщение об ошибке.

- Если после проведенного на предыдущем шаге сокращения кандидатов список все еще содержит элементы, среди них производится поиск наилучшего кандидата. Если список содержит только один элемент, этот элемент и выбирается в качестве наилучшего кандидата. В противном случае в этом списке производится поиск реализации функционального члена, набор формальных параметров которого наилучшим образом соответствует списку аргументов вызываемого функционального члена. Считается, что набор формальных параметров кандидата наилучшим образом среди всех кандидатов соответствует списку аргументов вызываемого функционального члена, если:
  - для каждого аргумента вызываемого функционального члена преобразование к типу соответствующего формального параметра кандидата не хуже, чем для любого другого кандидата;
  - по сравнению с любым другим кандидатом, по крайней мере для одного аргумента вызываемого функционального члена преобразование к типу соответствующего формального параметра кандидата лучше, чем у его конкурента.
- Если в результате этой операции возникает неоднозначность, не позволяющая выделить единого наилучшего кандидата, выдается сообщение об ошибке.

При сравнении преобразований типов формальных параметров используются следующие правила, позволяющие выявить лучшее преобразование:

- если оба формальных параметра имеют один и тот же тип, то ни одно из преобразований не может быть признано лучшим;
- в противном случае, если тип одного из формальных параметров совпадает с типом аргумента, из которого необходимо произвести преобразование, то преобразование к этому типу считается лучшим;
- если существует неявное преобразование из типа формального параметра первого кандидата в тип формального параметра второго кандидата и отсутствует обратное неявное преобразование, то преобразование типа формального параметра первого кандидата считается лучшим;

- преобразование к типу `sbyte` считается лучше преобразования к типам `byte`, `ushort`, `uint` или `ulong`;
- преобразование к типу `short` считается лучше преобразования к типам `ushort`, `uint` или `ulong`;
- преобразование к типу `int` считается лучше преобразования к типам `uint` или `ulong`;
- преобразование к типу `long` считается лучше преобразования к типу `ulong`;
- если ни одно из предыдущих правил выбора лучшего преобразования типа не может быть применено, ни одно из используемых преобразований не может быть признано лучшим.

Вызов функциональных членов

Рассмотрим процесс вызова функциональных членов. Процедура вызова статических и нестатических функциональных членов существенно различается, поскольку статические функциональные члены принадлежат всему типу и разделяются его объектами, а нестатические члены принадлежат только своему объекту типа. Следует помнить, что не бывает статических процедур доступа к индексатору и статические функциональные члены не могут быть виртуальными. После определения наилучшего кандидата на реализацию вызванного статического функционального члена не производятся никакие дополнительные действия, а просто вызывается найденная реализация.

Нестатические функциональные члены принадлежат только своему объекту типа, поэтому перегруженные операции не могут быть нестатическими. Нестатические функциональные члены могут быть как обычными, так и виртуальными. При вызове нестатического функционального члена недостаточно найти его реализацию, необходимо также определить объект типа, которому принадлежит вызываемый функциональный член. Процедуры вызова функциональных членов зависят от того, является ли их тип типом значения или типом ссылки:

- Если функциональный член принадлежит типу значения:
  - вычисляется значение выражения, определяющего объект типа, которому принадлежит данный функциональный член. Если в процессе вычисления данного выражения вызывается исключение, последующие действия не производятся;
  - если данное выражение не является переменной, создается временная локальная переменная соответствующего типа, которой присваивается вычисленное значение выражения. Исходный тип выражения переопределяется как ссылка на временную локальную переменную. Единственным способом доступа к этой локальной переменной является использование ключевого слова `this` в пределах вызываемого

функционального члена. Поэтому, если выражение, определяющее объект типа, не является переменной, вызывающая функция не может отследить изменения, внесенные в объект при вызове данного функционального члена;

- производится поиск наилучшей реализации данного функционального члена в соответствии с описанной выше процедурой;
- управление передается найденной реализации. При этом в переменную `this` записывается ссылка на переменную, содержащую объект класса, которому принадлежит вызываемый функциональный член.

❒ Если функциональный член принадлежит типу ссылки:

- вычисляется значение выражения, определяющего объект типа, которому принадлежит данный функциональный член. Если в процессе вычисления данного выражения вызывается исключение, последующие действия не производятся;
- производится поиск наилучшей реализации данного функционального члена в соответствии с описанной выше процедурой;
- если выражение, определяющее объект типа, имеет тип значения, этот тип преобразуется в тип `object`, и в дальнейшем производится работа с функциональными членами класса `object`;
- проверяется корректность вычисленного значения выражения. Если в результате вычислений получилось значение `null`, то вызывается исключение `NullReferenceException`, и процедура вызова функционального члена завершается;
- производится поиск вызываемой реализации: если вызываемый функциональный член не является виртуальным, то вызывается найденная на предыдущем шаге наилучшая реализация. В противном случае производится поиск реализации данного функционального члена в базовых классах и интерфейсах его типа;
- управление передается найденной реализации. При этом в переменную `this` записывается ссылка на объект, которому принадлежит вызываемый функциональный член.

В рассмотренной выше процедуре было упомянуто преобразование типа значения в тип ссылки. Эта операция производится в следующих случаях:

❒ если вызываемый функциональный член перегружает метод, унаследованный от типа `object`, и его вызов производится с использованием выражения, имеющего тип `object`;

❒ если вызываемый функциональный член является реализацией функционального члена интерфейса, и его вызов производится с использованием выражения, имеющего тип интерфейса;

❒ при вызове функционального члена с использованием делегата.

Во всех этих случаях считается, что создаваемый объект содержит переменную, имеющую тип значения, на которую ссылается переменная `this` в процессе выполнения данного функционального члена. Это, помимо прочего, означает, что при подобном вызове функциональный член может менять значение вызвавшего его выражения.

Первичные выражения

К *первичным выражениям* относятся простейшие выражения, из которых могут быть сформированы более сложные выражения. Некоторые первичные выражения могут включать в себя другие выражения. К первичным выражениям относятся:

- литералы;
- простые имена;
- выражение в скобках;
- доступ к члену;
- вызов функционального члена;
- доступ к элементу;
- ключевое слово `this`;
- ключевое слово `base`;
- постфиксное инкрементирование;
- постфиксное декрементирование;
- операция `new`;
- операция `typeof`;
- операция `sizeof`;
- установка режима проверки на переполнение;
- сброс режима проверки на переполнение.

Литералы

В C# *литералы* рассматриваются как значения, тип этого значения определяется самим литералом:

- *Логический литерал* имеет тип `bool` и может принимать два значения: `true` и `false`.
- *Целочисленный литерал* может иметь один из следующих типов: `int`, `uint`, `long` или `ulong`. Тип литерала определяется его значением и наличием или отсутствием суффикса типа. Этот тип литерала может быть десятичным и шестнадцатеричным.

- *Литерал формата с плавающей точкой* может иметь один из следующих типов: `float`, `double` или `decimal`. Тип литерала определяется его значением и наличием или отсутствием суффикса типа.
- *Символьные литералы* имеют тип `char`. Этот тип литерала представляет собой символ, заключенный в одиночные кавычки. В качестве символов могут выступать любые символы, кроме символов одиночной кавычки ('), обратной наклонной черты (\) и любого пробела, кроме простого пробела ( ), простые эскейп-последовательности, шестнадцатеричные эскейп-последовательности, представляющие собой эскейп-последовательность `\x`, за которой следует последовательность шестнадцатеричных цифр, и эскейп-последовательности Unicode.
- *Строковые литералы* имеют тип `string`. В C# могут использоваться два типа строковых литералов: *регулярные* и *дословные*. Регулярные строковые литералы представляют собой последовательность символов и простых или шестнадцатеричных эскейп-последовательностей, заключенных в двойные кавычки. Дословный строковый литерал состоит из символа `@`, за которым следует последовательность символов, заключенная в двойные кавычки. Основное отличие дословного строкового литерала от регулярного состоит в том, что в нем все символы воспринимаются "дословно", то есть не подвергаются преобразованию. Единственным исключением из этого правила являются двойные кавычки. Таким образом, подобные строковые литералы могут содержать фрагменты, совпадающие с простыми и шестнадцатеричными эскейп-последовательностями, которые не будут преобразовываться при работе с данным литералом.

Простые имена

Простое имя состоит из идентификатора. Соответствие между простым именем и обозначаемым им элементом устанавливается следующим образом:

- Если простое имя используется в блоке, блок содержит одноименную локальную переменную или аргумент, тогда считается, что простое имя используется для доступа к этой локальной переменной или аргументу и классифицируется как переменная.
- В противном случае для каждого типа, начиная с типа класса, структуры или объявления перечислимого типа, в котором использовано данное простое имя, и продолжая во всех классах или структурах, включающих в себя объявления уже просмотренных типов (если таковые имеются), производится поиск одноименных членов по следующим критериям:
  - если тип, в котором производится поиск, содержит один или несколько одноименных методов, то данному простому имени сопоставляется найденная группа методов, связанная с данным объектом класса или структуры;

- если тип, в котором производится поиск, содержит одноименный член, на который имеется ссылка в блоке конструктора, нестатическом методе или в нестатической процедуре доступа, то доступ к данному члену типа осуществляется с использованием полного имени, в котором в качестве выражения, определяющего объект класса или структуры, используется ключевое слово `this`;
- в противном случае доступ к данному члену типа осуществляется как к статическому члену класса.

❒ Если описанные выше процедуры не привели к успеху, то для всех пространств имен, включающих данное простое имя, вплоть до глобального пространства имен, производится поиск одноименных членов по следующим критериям:

- если пространство имен, в котором производится поиск, содержит одноименный член, то данному простому имени сопоставляется найденный член, который в зависимости от контекста классифицируется как пространство имен или тип;
- если произведенный до этого поиск не привел к успеху и объявление пространства имен содержит директиву определения альтернативного имени, связывающую данное имя с импортированным пространством имен или классом, тогда простому имени сопоставляется это пространство имен или тип;
- если произведенный до этого поиск не привел к успеху и во всех пространствах имен, импортированных соответствующими директивами, содержится только одно объявление одноименного типа, тогда простому имени сопоставляется этот тип. Если же и во всех пространствах имен, импортированных соответствующими директивами, содержится более одного объявления одноименного типа, то возникает неоднозначность и выдается сообщение об ошибке.

❒ Если все вышеописанные действия не позволили установить соответствие между простым типом и обозначаемым им элементом, простое имя считается неопределенным и выдается сообщение об ошибке.

Если в некотором блоке использовано простое имя, то во всех внутренних блоках этого блока данное простое имя не может быть переопределено для соответствия другому элементу. Это правило позволяет избежать неоднозначности в трактовке простого имени. Приведенный ниже пример содержит ошибку, поскольку идентификатор `d` использован в блоке метода `Some_Func` и переопределен во вложенном блоке условного оператора:

```
class Demo
{
        float d;
```

```
        void Some_Func(int i)
        {
                d = 1f;
                if( i > 0 )
                {
                        byte d = 1;    // Ошибка
                }
        }
}
```

Для того чтобы исправить эту ошибку, достаточно поместить обращения к идентификатору в разных вложенных блоках одного уровня, как это сделано в следующем примере:

```
class Demo
{
        float d;
        void Some_Func(int i)
        {
                if( i > 0 )
                {
                        byte d = 1;
                }
                else
                {
                        d = 1f;
                }
        }
}
```

Здесь следует обратить внимание, что важно не то, где определен идентификатор, а где он используется.

Описанное выше правило действует только в отношении простых имен. Поэтому приведенный выше пример может быть исправлен и таким образом:

```
class Demo
{
        float d;
        void Some_Func(int i)
        {
                this.d = 1f;
```

```
                if( i > 0 )
                {
                        byte d = 1;    // Ошибка
                }
        }
}
```

Эта возможность совместного использования одноименных идентификаторов часто используется в программах инициализации и в конструкторах:

```
class Demo
{
        float d;
        public Demo()
        {
                d = 0f;
        }
        public Demo(float d)
        {
                this.d = d;
        }
}
```

Выражение в скобках

Выражение в скобках, как явствует из его названия, представляет собой выражение, заключенное в круглые скобки. Его значение совпадает со значением выражения, заключенного в круглые скобки, и используется для изменения последовательности выполнения операций. Если выражение, заключенное в круглые скобки, представляет собой пространство имен, тип или группу методов, то компилятор выдает сообщение об ошибке.

Доступ к члену

Выражение, осуществляющее доступ к члену типа представляет собой первичное выражение или предопределенный тип, отделенный точкой от идентификатора, определяющего член, к которому осуществляется доступ. Это выражение часто называют полным именем члена типа. Предопределенными являются следующие типы: `bool`, `byte`, `char`, `decimal`, `double`, `float`, `int`, `long`, `object`, `sbyte`, `short`, `string`, `uint`, `ulong` и `ushort`.

Поиск члена, к которому осуществляется доступ, производится следующим образом:

- Если в качестве первичного выражения используется идентификатор пространства имен, а член, к которому осуществляется доступ, представляет собой имя доступного члена данного пространства имен, то доступ осуществляется к этому члену, который в зависимости от контекста рассматривается как пространство имен или тип.
- Доступ осуществляется к члену типа, идентифицируемого первичным выражением, определяющим тип, или предопределенным типом, и данный тип содержит член с искомым именем, то результат поиска определяется следующим образом:
  - если член представляет собой тип, то результатом поиска является этот тип;
  - если члену соответствуют один или несколько методов, то результатом поиска является группа методов, не привязанная ни к какому объекту типа;
  - если члену соответствует статическое свойство, то результатом поиска является процедура доступа к свойству, не привязанная ни к какому объекту типа;
  - если члену соответствует статическое поле, то, если это поле имеет модификатор `readonly` и ссылка на это поле производится из статического конструктора класса или структуры, в которой определено данное поле, то результатом поиска является значение соответствующего статического поля. В противном случае результатом является переменная, соответствующая данному статическому полю;
  - если члену соответствует статическое событие, то, если ссылка на это событие производится в пределах класса или структуры, в которой определено данное событие, то обработка этого события производится так же, как если бы оно было статическим полем или свойством. В противном случае результатом является процедура доступа к событию, не привязанная ни к какому объекту типа;
  - если члену соответствует константа, то результатом поиска является значение этой константы;
  - если члену соответствует переменная перечислимого типа, то результатом поиска является значение этой переменной перечислимого типа;
  - если к члену неприменимо ни одно из перечисленных выше правил, то ссылка на член считается некорректной и выдается сообщение об ошибке.

- Если в качестве первичного выражения выступает процедура доступа к свойству или индексатору, переменная или значение, имеющие определенный тип, и этот тип содержит член с искомым именем, то результат поиска определяется следующим образом:
  - если в качестве первичного выражения выступает процедура доступа к свойству или индексатору, то, прежде чем приступать к последующим операциям, вызывается процедура доступа к данному свойству или индексатору, позволяющая определить значение, которое тут же присваивается временной локальной переменной;
  - если члену соответствуют один или несколько методов, то результатом поиска является группа методов, связанная с объектом, идентифицируемым первичным выражением;
  - если члену соответствует нестатическое свойство, то результатом поиска является процедура доступа к свойству, связанная с объектом, идентифицируемым первичным выражением;
  - если первичное выражение имеет тип класса, а члену соответствует нестатическое поле данного класса, то:
    - ⇒ если первичное выражение имеет значение `null`, вызывается исключение `NullReferenceException`;
    - ⇒ в противном случае, если это поле имеет модификатор `readonly` и ссылка на это поле производится из конструктора объекта класса, в котором определено данное поле, то результатом поиска является значение поля объекта структуры, идентифицируемого первичным выражением;
    - ⇒ если первичное выражение имеет ненулевое значение и поле не имеет модификатора `readonly`, то результатом поиска является переменная, соответствующая полю объекта структуры, идентифицируемого первичным выражением.
  - если первичное выражение имеет тип структуры, а члену соответствует нестатическое поле данного класса, то:
    - ⇒ если это поле имеет модификатор readonly и ссылка на это поле производится из конструктора объекта структуры, в которой определено данное поле, то результатом поиска является значение поля объекта структуры, идентифицируемого первичным выражением;
    - ⇒ в противном случае результатом поиска является переменная, соответствующая полю объекта структуры, идентифицируемого первичным выражением.
  - если члену соответствует нестатическое событие, то, если ссылка на это событие производится в пределах класса или структуры, в которой определено данное событие, то обработка этого события производится так же, как если бы оно было нестатическим полем или свойством.

В противном случае результатом является процедура доступа к событию, связанная с объектом, идентифицируемым первичным выражением;

- если к члену неприменимо ни одно из перечисленных выше правил, то ссылка на член считается некорректной и выдается сообщение об ошибке.

Использование типов как простых имен

Поскольку синтаксис применения типов и простых имен существенно различается, в C# допускается использовать один и тот же идентификатор для обозначения типов и простых имен, если последние применяются для обращения к константам, полям, свойствам, локальным переменным, имеющим тип, совпадающий с их именем. В этом случае не может возникнуть никакой неоднозначности, поскольку это правило просто позволяет использовать имя типа для вызова его нестатических членов.

Рассмотрим это на примере:

```
class Demo
{
        public static float d;
        public void Some_Func(float f)
        {
                d = f;
        }
}

class Test
{
        public Demo   Demo = new Demo();

        void Other_Func()
        {
                Demo.d = 1f;          // Вызов статического поля
                Demo.Some_Func( 1f ); // Вызов нестатического метода
        }
}
```

Вызов функционального члена

Вызов функционального члена имеет следующий синтаксис: первичное выражение и следующие за ним скобки, в которых может помещаться список аргументов. В качестве первичного выражения при вызове функционального члена может использоваться группа методов или значение, имеющее тип

делегата. Список аргументов при вызове функционального члена содержит значения или ссылки на переменные, передаваемые формальным параметрам данного метода.

Если вызываемый метод или делегат возвращает значение типа `void`, то значение соответствующего выражения не определено. Выражение с неопределенным значением не может выступать в качестве операнда любой операции и может использоваться только как оператор-выражение. В противном случае результатом данного выражения является значение, возвращаемое методом или делегатом.

Вызов метода

Вызов метода является частным случаем вызова функционального члена, в котором в качестве первичного выражения используется группа методов, представляющая собой один вызываемый метод или группу, из которой следует выбрать один вызываемый метод. Выбор метода производится на основании типов элементов списка аргументов вызываемого типа. Этот выбор производится в процессе компиляции приложения.

Выбор вызываемого метода в группе методов производится следующим образом:

- Создается список кандидатов на реализацию вызванного метода. После чего из этого списка исключаются недопустимые методы. При этом используются следующие правила:
  - из списка исключаются все методы, наборы формальных параметров которых не соответствуют списку аргументов вызываемого метода;
  - если в некотором классе найден метод, набор формальных параметров которого соответствует списку аргументов вызываемого метода, то из списка исключаются все методы его базовых классов.
- Если после проведения описанной выше процедуры список кандидатов на реализацию вызванного метода пуст, то данный метод не имеет реализации, о чем выводится соответствующее сообщение об ошибке. Если методы в списке кандидатов принадлежат различным типам, при выборе реализации возникает неоднозначность, о чем выводится соответствующее сообщение об ошибке (эта ситуация может возникнуть только при вызове метода интерфейса, использующего множественное наследование).
- Для поиска наилучшей реализации метода используются правила, описанные в предыдущем разделе. Если эти правила не позволяют выделить одну наилучшую реализацию, возникает неоднозначность, о чем выводится соответствующее сообщение об ошибке.
- Для найденной наилучшей реализации вызванного метода производится проверка допустимости его вызова. Если найденная реализация является статическим методом, то для его вызова должно быть использовано простое имя или доступ к члену с указанием типа. Если найденная реализа-

ция не является статическим методом, то для его вызова должно быть использовано простое имя, доступ к члену с указанием переменной или значения или же доступ к скрытому методу базового класса. Если же эти условия не выполняются, компилятор выдает сообщение об ошибке.

- Метод, выбранный и проверенный, в соответствии с описанной выше процедурой вызывается в соответствии с описанными в предыдущем разделе правилами вызова функциональных членов.

Применение описанных выше правил позволяет найти, по крайней мере одну реализацию метода в цепи наследования, начинающейся с типа, объявленного при вызове, и вызвать его, используя правила вызова функциональных членов.

Вызов делегата

Вызов делегата является частным случаем вызова функционального члена, в котором в качестве первичного выражения используется значение, имеющее тип делегата. Кроме того, список аргументов типа делегата должен соответствовать списку аргументов вызываемого функционального члена.

Вызов делегата производится следующим образом:

- Определяется значение первичного выражения, имеющего тип делегата. Если при определении этого значения вызывается исключение, то последующие операции не производятся.
- Проверяется корректность полученного значения первичного выражения. Если получено значение `null`, вызывается исключение `NullReferenceException` и никакие последующие операции не производятся.
- Если все предыдущие проверки прошли успешно, полученное значение преобразуется к типу ссылки на объект делегата. После этого производится вызов функционального члена, на который ссылается делегат. Если этот метод не является статическим, объект, для которого вызывается данный функциональный член, совпадает с объектом, на который ссылается делегат.

Доступ к элементу

Доступ к элементу имеет следующий синтаксис: первичное выражение, за которым следует список выражений, заключенный в квадратные скобки. Элементы списка выражений отделяются друг от друга запятой.

Если первичное выражение при доступе к элементу имеет тип массива, то доступ осуществляется к элементу массива. В противном случае первичное выражение должно иметь тип класса, структуры или интерфейса, в которых должен быть определен, по крайней мере, один индексатор, имеющий список выражений, совместимый со списком выражений, по которому осуществляется доступ.

Доступ к элементу массива

Как уже говорилось выше, при доступе к элементу массива первичное выражение должно иметь тип массива. Число выражений в списке выражений должно совпадать с рангом массива, а каждое из этих выражений должно иметь тип `int` или тип, имеющий неявное преобразование к типу `int`.

Результатом доступа к элементу массива является переменная, имеющая тип элемента массива, и представляющая собой элемент массива, выбранный в соответствии со значениями элементов списка выражений.

Доступ к элементу массива производится следующим образом:

- Определяется значение первичного выражения. Если при определении этого значения вызывается исключение, то последующие операции не производятся.
- Вычисляются значения элементов списка выражений в порядке слева направо. Каждое вычисленное значение сразу же преобразуется к типу `int`. Если в процессе вычисления значений выражений или при преобразовании полученных значений к типу `int` вызывается исключение, то последующие операции не производятся.
- Проверяется корректность полученного значения первичного выражения. Если получено значение `null`, вызывается исключение `NullReferenceException` и никакие последующие операции не производятся;
- Если значение какого-либо выражения в списке выражений выходят за диапазон, определенный при создании данного объекта типа массива, вызывается исключение `IndexOutOfRangeException` и никакие последующие операции не производятся.
- В соответствии со значениями элементов списка выражений определяется положение элемента массива, к которому осуществляется доступ, и этот элемент считается результатом проведенного поиска.

Доступ по индексатору

Как уже говорилось выше, при доступе по индексатору первичное выражение должно иметь тип класса, структуры или интерфейса, в которых должен быть определен, по крайней мере один индексатор, имеющий список выражений, совместимый со списком выражений, по которому осуществляется доступ.

Доступ по индексатору осуществляется следующим образом:

- строится список индексаторов того типа, для которого осуществляется вызов. В этот список включаются все индексаторы, объявленные для данного типа и в его базовых типах, не содержащие модификатора `override` и доступные в данном контексте.

- Из полученного списка исключаются все неприменимые или скрытые индексаторы. При этом действуют следующие правила:
  - из списка исключаются все индексаторы, список выражений которых не совместим со списком выражений вызываемого индексатора;
  - если в каком-либо типе найден индексатор, список выражений которого совместим со списком выражений вызываемого индексатора, то из списка удаляются все индексаторы, определенные в его базовом типе.
- Если после проведения описанной выше процедуры список кандидатов на реализацию вызванного индексатора пуст, то данный индексатор не имеет реализации, о чем выводится соответствующее сообщение об ошибке. Если индексаторы в списке кандидатов принадлежат различным типам, при выборе реализации возникает неоднозначность, о чем выводится соответствующее сообщение об ошибке (эта ситуация может возникнуть только при вызове индексатора интерфейса, использующего множественное наследование).
- Для поиска наилучшей реализации индексатора используются правила, описанные в предыдущем разделе. Если эти правила не позволяют выделить одну наилучшую реализацию, возникает неоднозначность, о чем выводится соответствующее сообщение об ошибке.
- В результате выполнения описанных выше операций получается ссылка на найденную реализацию индексатора, связанную с объектом, для которого вызывался данный индексатор, и списком аргументов, совместимым со списком выражений вызванного индексатора.

В зависимости от контекста вызова индексатора в результате проведенных выше операций вызывается процедура доступа `get` или процедура доступа `set` данного индексатора. Процедура доступа `set` вызывается при вызове индексатора справа от операции присваивания, а процедура доступа `get` вызывается для получения его текущего значения.

Доступ к символу строки

В классе `string` определен индексатор, позволяющий получить доступ к любому ее символу. Этот индексатор объявлен следующим образом:

```
public char this[int index] { get; }
```

Такое объявление означает, что данный индексатор может использоваться только для чтения, имеет единственный аргумент типа `int` и возвращает элемент типа `char`. Значение, передаваемое аргументу `index`, должно быть не меньше нуля и меньше числа символов в строке.

Ключевое слово this

Ключевое слово `this` может использоваться только в блоке конструктора, нестатического метода или нестатической процедуры доступа. Оно имеет одно из следующих значений:

- При использовании в составе первичного выражения, расположенного в конструкторе класса, оно представляет собой значение ссылки на создаваемый объект. Типом этого значения является тип класса, в конструкторе которого производится ссылка.
- При использовании в составе первичного выражения, расположенного в нестатическом методе или в нестатической процедуре доступа класса, оно представляет собой значение ссылки на объект, для которого вызывается данный метод или процедура доступа. Типом этого значения является тип класса, в котором производится ссылка.
- При использовании в составе первичного выражения, расположенного в конструкторе структуры, оно представляет собой значение создаваемого объекта структуры. Типом этого значения является тип структуры, в конструкторе которого производится ссылка. В данном случае переменная `this` эквивалентна аргументу, имеющему тип структуры и модификатор `out`. Это означает, прежде всего, то, что эта переменная должна быть явным образом инициализирована в каждой ветви исполнения конструктора.
- При использовании в составе первичного выражения, расположенного в нестатическом методе или в нестатической процедуре доступа структуры, оно представляет собой значение объекта структуры, для которого вызывается данный метод или процедура доступа. Типом этого значения является тип структуры, в котором производится ссылка. В данном случае переменная `this` эквивалентна аргументу, имеющему тип структуры и модификатор `ref`.

Использование ключевого слова `this` в первичных выражениях в любом другом контексте, кроме одного из перечисленных выше, является ошибкой. Особо следует помнить, что недопустимо использование ключевого слова `this` в статических методах, статических процедурах доступа или в инициализаторах полей.

Ключевое слово base

Ключевое слово `base` может использоваться для доступа к члену или для доступа по индексатору. Поэтому за этим ключевым словом может стоять точка, отделяющая его от простого имени члена, или квадратные скобки, содержащие список выражений.

Ключевое слово `base` используется для доступа к членам базового класса, скрытым одноименными членами в данном классе или структуре. Ключевое

слово base может использоваться только в блоке конструктора, нестатического метода или нестатической процедуры доступа. Выражение base.Identifier используется для доступа к члену Identifier базового класса, того класса или структуры, в которой располагается данное выражение. Аналогично выражение base[Expressions] используется для доступа к соответствующему индексатору базового класса.

В процессе компиляции выражение base.Identifier интерпретируется как выражение ((Base_Class) this).Identifier, а выражение base[Expressions] — как выражение((Base_Class) this)[Expressions], где Base_Class — идентификатор базового класса.

Если ключевое слово base используется для доступа к функциональному члену (методу или процедуре доступа), этот член не считается виртуальным. Поэтому ключевое слово base может использоваться для доступа к перегружаемой версии при перегрузке функционального члена. Это бывает полезным в том случае, когда при перегрузке добавляются новые функциональные возможности, не затрагивающие работу перегружаемой версии. Ключевое слово base не может использоваться для доступа к абстрактным функциональным членам.

Постфиксное инкрементирование и декрементирование

Выражение постфиксного инкрементирования и декрементирования представляет собой первичное выражение, за которым следует знак операции инкрементирования (++) или декрементирования (--). В качестве первичного выражения может выступать переменная или процедура доступа к свойству или индексатору. Результат выражения постфиксного инкрементирования или декрементирования имеет тот же тип, что и операнд.

Если в качестве первичного выражения выступает процедура доступа к свойству или индексатору, то соответствующее свойство или индексатор должны иметь процедуры доступа как для чтения, так и для записи. Если в свойстве или индексаторе отсутствует процедура доступа get или set, компилятор выдает сообщение об ошибке.

Типы sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal и все перечислимые типы содержат предопределенные операции инкрементирования и декрементирования. Предопределенная операция инкрементирования добавляет 1 к своему аргументу, а предопределенная операция инкрементирования вычитает 1 из своего аргумента.

В процессе выполнения приложения выражения постфиксного инкрементирования и декрементирования исполняются следующим образом:

- Если в качестве первичного выражения выступает переменная:
  - вычисляется значение первичного выражения для получения доступа к переменной;

- значение этой переменной сохраняется;
- для сохраненной переменной вызывается соответствующая операция;
- новое значение сохраненной переменной записывается в область памяти, выделенную для данной переменной;
- значением выражения постфиксного инкремента или декремента является результат выполненной операции.

❒ Если в качестве первичного выражения выступает процедура доступа:

- если процедура доступа не является статической, определяется объект, для которого вызывается данная процедура доступа;
- если вызывается процедура доступа к индексатору, вычисляются значения выражений в его списке выражений;
- вызывается соответствующая процедура доступа `get`, и во временной локальной переменной сохраняется возвращенное ей значение;
- для временной локальной переменной вызывается соответствующая операция;
- вызывается соответствующая процедура доступа `set`, которой в качестве аргумента `value` передается новое значение временной локальной переменной;
- значением выражения постфиксного инкремента или декремента является результат выполненной операции.

Предопределенные операции инкрементирования и декрементирования могут вызываться как в префиксной, так и в постфиксной формах. Результат операций инкрементирования и декрементирования в постфиксной форме соответствует значению операнда **до** выполнения операции, а результат этих операций в префиксной форме соответствует значению операнда **после** выполнения операции.

Перегруженные операции инкрементирования и декрементирования также могут вызываться в префиксной и в постфиксной формах, но при этом отсутствует возможность задать различные реализации операции для каждой из этих форм. Поэтому результат перегруженной операции инкрементирования или декрементирования не зависит от того, в какой форме она была вызвана.

Операция new

Операция `new` может использоваться для создания объектов, массивов и делегатов. Она не только позволяет создать экземпляр требуемого типа, но и выделить для него динамическую область оперативной памяти, если в этом возникает необходимость.

Создание объекта

Выражение, создающее объект, имеет следующий синтаксис: ключевое слово `new` и идентификатор типа, за которым следуют круглые скобки, в которых может размещаться список аргументов. Данное выражение позволяет создавать объекты, имеющие тип значения, класса или структуры. Используемый тип класса не может быть абстрактным. Список аргументов может присутствовать только в выражении, создающем объект класса или структуры.

Создание объекта происходит в два этапа. На первом этапе компилятор определяет значение, возвращаемое данным выражением, а на втором этапе, в процессе выполнения приложения для объектов выделяется динамическая область памяти.

При создании объекта компилятором производятся следующие действия:

- Если создается объект типа значения, то для него вызывается используемый по умолчанию конструктор. Поскольку этот конструктор не имеет пустого набора формальных параметров, то и список аргументов в данном выражении должен быть пуст. Данное выражение имеет тип, определяемый идентификатором типа, и значение, задаваемое конструктором, используемым по умолчанию.
- Если создается объект класса или структуры:
  - при попытке создания объекта абстрактного класса компилятор выдает сообщение об ошибке;
  - составляется список кандидатов на реализацию вызываемого конструктора, который должен содержать все конструкторы, объявленные в данном классе. Если список кандидатов пуст или правила, используемые при поиске реализации метода, не позволяют определить единственного наилучшего кандидата, компилятор выдает сообщение об ошибке;
  - данное выражение имеет тип, определяемый идентификатором типа, и значение, задаваемое конструктором, выбранным по описанным выше правилам.
- Если описанные выше правила не позволили найти реализацию конструктора, компилятор выдает сообщение об ошибке.

В процессе выполнения приложения для каждого создаваемого объекта необходимо выделить память. Если объект имеет тип значения, то он занимает ту область памяти, которая выделена ему компилятором, и для него не нужно выделять дополнительную память. Если же объект имеет тип ссылки, к которому относятся типы класса и структуры, то в области памяти, выделенной компилятором, хранится только ссылка на объект, а для хранения самого объекта должна быть выделена динамическая память.

При создании объекта класса или структуры в процессе выполнения приложения выполняются следующие действия:

- Если создается объект класса:
  - выделяется память под новый объект класса. Если свободной памяти недостаточно для размещения в ней создаваемого объекта, вызывается исключение `OutOfMemoryException` и последующие действия не производятся;
  - все поля нового объекта класса инициализируются своими значениями по умолчанию;
  - вызывается выбранный компилятором конструктор класса, которому автоматически передается ссылка на созданный объект. Доступ к этому объекту класса в пределах конструктора может быть осуществлен с использованием ключевого слова `this`.
- Если создается объект структуры:
  - создается новый объект структуры и сохраняется во временной локальной переменной. Поскольку конструктор структуры должен в явном виде присвоить значения каждому полю созданного объекта, нет никакой необходимости инициализировать временную локальную переменную;
  - вызывается выбранный компилятором конструктор структуры, которому автоматически передается ссылка на созданный объект. Доступ к этому объекту структуры в пределах конструктора может быть осуществлен с использованием ключевого слова `this`.

Создание массива

Создание массива может быть произведено с использованием одной из двух форм. Первая из них имеет следующий синтаксис: ключевое слово `new` и идентификатор типа, не являющегося типом массива, за которым расположен список выражений, помещенный в квадратные скобки. Элементы списка выражений отделяются друг от друга запятой. Все выражения в этом списке должны иметь тип `int` или тип, имеющий неявное преобразование к типу `int`. Значения этих выражений определяют диапазон возможных значений соответствующей размерности создаваемого массива. За закрывающей квадратной скобкой могут располагаться спецификаторы ранга и инициализатор массива. Вторая форма создания массива имеет следующий синтаксис: ключевое слово `new` и идентификатор типа, являющегося типом массива, за которым расположен его инициализатор.

При создании массива с использованием первой формы данного выражения создается массив, тип которого можно получить, исключив выражения из списка, оставив только разделяющие их запятые. Например, выражение `new`

`bool[3, 4]` создает массив типа `bool[,]`, а выражение `new bool[5][,]` создает массив типа `bool[][,]`. Если данное выражение содержит инициализатор массива, то все выражения в его списке выражений должны быть константными, а ранг и диапазон возможных значений каждой размерности создаваемого массива должен соответствовать рангу и диапазону возможных значений соответствующей размерности его инициализатора.

При создании массива с использованием второй формы данного выражения ранг создаваемого массива должен соответствовать рангу его инициализатора, а диапазон возможных значений каждой размерности создаваемого массива определяется диапазоном возможных значений соответствующей размерности его инициализатора. Таким образом, выражение

```
new int[,] {{0, 1, 2}, {3, 4, 5}};
```

соответствует выражению

```
new int[2, 3] {{0, 1, 2}, {3, 4, 5}};
```

Данное выражение имеет значение ссылки на создаваемый экземпляр массива.

При создании массива в процессе выполнения приложения выполняются следующие действия:

- В порядке слева направо вычисляются значения выражений в списке, определяющем диапазон возможных значений каждой размерности создаваемого массива. После вычисления значения каждого выражения производится его преобразование к типу `int`. Если при вычислении значения выражения или в процессе преобразования его типа вызывается исключение, то никакие последующие действия не производятся.
- Производится проверка полученных значений границ диапазона. Если хотя бы одно из полученных значений оказывается меньшим нуля, вызывается исключение `IndexOutOfRangeException` и последующие действия не производятся.
- Выделяется память под массив соответствующего размера. Если свободной памяти недостаточно для размещения в ней создаваемого массива, вызывается исключение `OutOfMemoryException` и последующие действия не производятся.
- Все элементы создаваемого массива инициализируются своими значениями по умолчанию.
- Если массив содержит инициализатор, тогда последовательно производится вычисление значений выражений, помещенных в инициализатор массива, и присваивание этих значений соответствующим элементам созданного массива. Вычисление значений и присваивание производится в порядке появления выражений в инициализаторе массива. То есть элементы инициализируются в порядке возрастания значения индекса. В многомерных массивах индексы возрастают справа налево: чем правее расположен ин-

декс, тем чаще он изменяется в процессе инициализации. Если при вычислении значения выражения или в процессе присваивания полученного значения элементу массива вызывается исключение, то инициализация заканчивается и все остальные элементы массива остаются неинициализированными (то есть они содержат свои значения по умолчанию).

Выражение создания массива не может одновременно создать несколько массивов. Например, следующее выражение содержит ошибку:

```
new short[10][20];    // Ошибка
```

Это связано с тем, что в данном случае определен массив из десяти элементов, имеющих тип `short[]`. Таким образом, в данном выражении предпринята попытка создать сразу 11 массивов (один для хранения элементов типа `short[]` и десять — для хранения элементов типа `short`).

Для создания всех этих массивов может быть использован следующий фрагмент программы:

```
short[][] a = new short[10][];
for (int i = 0; i < 10; i++)
       a[i] = new short[20];
```

В данном случае все массивы создаются по отдельности. Данное правило действует только в том случае, если создаются массивы, элементы которых имеют тип массива. Приведенный выше массив может быть создан и в следующем виде:

```
new short[10, 20];
```

Это выражение не содержит ошибки, поскольку в нем определен двумерный массив, все элементы которого имеют тип `short`.

Создание делегата

Выражение для создания делегата имеет следующий синтаксис: ключевое слово `new` и идентификатор типа, являющегося типом делегата, за которым расположен аргумент, помещенный в круглые скобки. Аргумент выражения для создания делегата представляет собой выражение, являющееся группой методов, или значение, имеющее тип делегата. Если аргументом данного выражения является группа нестатических методов, то аргумент, кроме того, определяет объект, для которого создается делегат. Если аргументом выражения для создания делегата является значение, имеющее тип делегата, то он определяет экземпляр делегата, для которого создается копия.

В процессе компиляции приложения выражение для создания делегата обрабатывается следующим образом:

- если аргументом данного выражения является группа методов:
  - Если для указания на данную группу методов использовано ключевое слово `base`, компилятор выдает сообщение об ошибке;

- набор методов, определяемый аргументом, должен содержать только один метод, набор формальных параметров и тип возвращаемого значения которого полностью идентичен создаваемому делегату. Именно на этот метод будет ссылаться создаваемый делегат. Если не существует ни одного такого метода, или если имеется несколько методов, отвечающих поставленным критериям, выдается сообщение об ошибке. Если выбран нестатический метод, то объект, для которого он вызывается, становится целевым объектом создаваемого делегата;
- как и в случае его вызова, выбранный метод должен быть совместим с контекстом группы методов. Если метод является статическим, доступ к группе методов должен производиться по простому имени или по полному имени, содержащему идентификатор типа. Если метод не является статическим, доступ к группе методов должен производиться по простому имени или по полному имени, содержащему значение или переменную соответствующего типа. Если выделенный метод не соответствует контексту группы методов, выводится сообщение об ошибке;
- результатом выражения является значение созданного делегата, ссылающегося на выбранный метод и целевой объект.

❒ Если аргументом данного выражения является выражение, имеющее тип делегата:

- это значение должно иметь тот же самый набор формальных параметров и тип возвращаемого значения, что и создаваемое значение типа делегата. В противном случае выдается сообщение об ошибке;
- результатом выражения является значение нового делегата, ссылающегося на тот же метод и целевой объект, что и его аргумент.

❒ Если описанные выше правила не могут быть применены, выражение считается некорректным и выдается сообщение об ошибке.

В процессе выполнения приложения выражение для создания делегата исполняется следующим образом:

❒ Если аргументом данного выражения является группа методов:

- Если выбранный метод является статическим, делегат имеет нулевой целевой объект. Если же этот метод не является статическим, производится анализ выбранного метода для определения связанного с ним объекта:

  ⇒ Определяется значение первичного выражения, определяющего объект, для которого вызывается данный метод. Если в процессе определения значения этого выражения вызывается исключение, никакие последующие действия не производятся;

⇒ если первичное выражение, значение которого определялось на предыдущем шаге, имеет тип ссылки, то вычисленное значение становится целевым объектом делегата. Если это выражение имеет значение null, вызывается исключение `NullReferenceException` и никакие последующие действия не производятся.

⇒ если первичное выражение имеет тип значения, то этот тип преобразуется к типу object и ссылка на полученный при этом объект становится целевым объектом делегата.

- выделяется память для нового делегата. Если свободной памяти недостаточно для размещения в ней создаваемого делегата, вызывается исключение `OutOfMemoryException`, и последующие действия не производятся;
- производится инициализация нового делегата ссылкой на метод, определенный в процессе компиляции, и на определенный выше целевой объект;

❒ Если аргументом данного выражения является выражение, имеющее тип делегата:

- определяется значение данного выражения. Если в процессе определения значения этого выражения вызывается исключение, никакие последующие действия не производятся;
- если это выражение имеет значение `null`, вызывается исключение `NullReferenceException`, и никакие последующие действия не производятся;
- выделяется память для нового делегата. Если свободной памяти недостаточно для размещения в ней создаваемого делегата, вызывается исключение `OutOfMemoryException`, и последующие действия не производятся;
- производится инициализация нового делегата ссылкой на тот же самый метод и объект, на который ссылается аргумент данного выражения.

Метод и объект, на который ссылается делегат, определяются в процессе его создания и остаются неизменными до уничтожения этого делегата. Нельзя создать делегата для конструктора, свойства, индексатора или перегруженной операции.

Рассмотрим пример создания делегата:

```
delegate int Int_Func(int i);
class Some_Class
{
        Int_Func Field = new Int_Func( Fact );
        static int Fact(int i)
        {
```

```
            int    k = 1;
            while( i-- > 0 )
                   k *= i;
            return k;
      }
      static long Fact(long i)
      {
            int    k = 1;
            while( i-- > 0 )
                   k *= i;
            return k;
      }
}
```

В данном примере поле `Some_Class.Field` инициализируется делегатом, ссылающимся на первый метод `Fact`, поскольку для него совпадают набор формальных параметров и тип возвращаемого значения делегата. Если бы описание этого метода отсутствовало, компилятор выдал бы сообщение об ошибке.

Операция typeof

Операция `typeof` используется для получения объекта `System.Type` для указанного типа. Данное выражение состоит из ключевого слова `typeof`, за которым в круглых скобках помещен идентификатор типа.

Рассмотрим пример использования этой операции:

```
class Demo
{
      static void Main()
      {
            Type[] t = {
                   typeof(int),
                   typeof(float[])
            };
            if( t[0].Name == "Int32" )
                   Console.WriteLine("OK");
            if( t[1].Name == "Single[]" )
                   Console.WriteLine("OK");
            if( typeof(string).Name == "String" )
                   Console.WriteLine("OK");
      }
}
```

В результате выполнения этой программы будут выведены три строки `OK`. Как видно из приведенного примера, имена типов не совпадают с их идентификаторами.

Операция sizeof

Операция `sizeof` используется для получения размера в байтах, необходимого для хранения значения указанного типа значения. Данное выражение состоит из ключевого слова `sizeof`, за которым в круглых скобках помещен идентификатор типа.

Это выражение может использоваться только в небезопасном режиме и только для типов значения. Работа в небезопасном режиме означает, что эта операция может вызываться только в небезопасных членах или в небезопасных блоках. Перегрузка данной операции запрещена. Следует также отметить, что в отличие от C++ в C# в качестве аргумента операции `sizeof` не может выступать переменная типа, по которому следует получить информацию.

Установка и сброс проверки на переполнение

Для установки и сброса режима проверки на переполнение используются операции `checked` и `unchecked` . Данное выражение состоит из ключевого слова `checked` или `unchecked` и выражения, помещенного в круглые скобки.

Операция `checked` устанавливает режим проверки на переполнение при выполнении операций над целыми числами в выражении, заключенном в круглые скобки. Операция `unchecked` сбрасывает этот режим в данном выражении. В остальном эти операции эквивалентны выражению в скобках.

Операции `checked` и `unchecked` проверяют или не проверяют результаты следующих операций:

- предопределенных операций `++` и `--`, если их операнды имеют один из целочисленных типов;
- предопределенной унарной операции `-`, если ее операнд имеет один из целочисленных типов;
- предопределенных бинарных операций `+`, `-`, `*` и `/`, если оба их операнда имеют один из целочисленных типов;
- явное преобразование типов из одного целочисленного типа в другой.

Если в результате выполнения любой из вышеперечисленных операций получается результат, выходящий за диапазон допустимых значений типа, в который он должен быть преобразован, операции `checked` и `unchecked` производят следующие действия:

- Если выражение, в котором произошло переполнение, является операндом операции `checked` и представляет собой константное выражение, то

компилятор выдает сообщение об ошибке. Если же переполнение возникло в процессе выполнения программы, операция `checked` вызывает исключение `OverflowException`.

- Если переполнение произошло в операнде операции `unchecked`, результат урезается путем отбрасывания старших бит результата операции, вызвавшей переполнение.

Если в процессе выполнения программы возникает переполнение в выражении, не являющимся операндом операций `checked` или `unchecked`, и не вызываемом в блоке соответствующих операторов, то его обработка зависит от внешних условий (ключей компилятора и конфигурации среды исполнения), однако, в любом случае это переполнение будет обработано по алгоритму, используемому операцией `checked`, или по алгоритму, используемому операцией `unchecked`.

Для константных выражений (вычисляемых в процессе компиляции) по умолчанию используется алгоритм операции `checked`. То есть, если операция или оператор `unchecked` явным образом не использованы, компилятор выдает сообщение об ошибке.

Операции `checked` и `unchecked` проверяют переполнение только в выражении, заключенном в круглые скобки. Если это выражение содержит вызов функции, то действие этих операций не распространяется на тело этих функций.

Одним из стандартных способов использования операции `unchecked` является преобразование шестнадцатеричных констант, имеющих по умолчанию тип `uint`, к типам целых чисел со знаком, как это продемонстрировано в приведенном ниже примере:

```
class Demo
{
        const short MinNeg = unchecked((short)0xFFFF); // MinNeg = -32768
        const short MaxNeg = unchecked((short)0x8000); // MaxNeg = -1
}
```

Унарные выражения

Унарные выражения включают в себя:

- первичное выражение;
- унарную операцию `+`;
- унарную операцию `-`;
- унарную операцию `!`;

- унарную операцию ~;
- унарную операцию *;
- унарную операцию &;
- префиксное инкрементирование;
- префиксное декрементирование;
- преобразование типов.

Унарная операция +

Унарная операция + имеет следующий синтаксис: знак операции +, за которым следует операнд. Поскольку эта операция может быть перегружена, то выполняемые ею действия зависят от ее конкретной реализации для типа операнда. Однако во всех случаях при реализации этой операции операнд преобразуется к типу аргумента выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

Ниже приведен список предопределенных операций +:

```
int     operator +(int x);
uint    operator +(uint x);
long    operator +(long x);
ulong   operator +(ulong x);
float   operator +(float x);
double  operator +(double x);
decimal operator +(decimal x);
```

Унарная операция

Унарная операция имеет следующий синтаксис: знак операции –, за которым следует операнд. Поскольку эта операция может быть перегружена, то выполняемые ею действия зависят от ее конкретной реализации для типа операнда. Однако во всех случаях при реализации этой операции операнд преобразуется к типу аргумента выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

Ниже приведен список предопределенных операций:

```
int             operator -(int x);
long            operator -(long x);
float           operator -(float x);
double          operator -(double x);
decimal operator -(decimal x);
```

Результатом выполнения данной операции над операндами типов является вычитание значения операнда из нулевого значения.

Если данная операция для операндов типа `int` или `long` производится в операции или операторе `checked` и ее операнд имеет значение минимального отрицательного числа данного типа, то вызывается исключение `OverflowException`. Если же данная операция для операндов типа `int` или `long` производится в операции или операторе `unchecked` и ее операнд имеет значение минимального отрицательного числа данного типа, то его значение не изменяется.

Если операнд данной операции имеет тип `uint`, то его значение преобразуется к типу `long` и результат операции также имеет тип `long`. Единственным исключением из этого правила является возможность преобразования десятичного целочисленного литерала –2147483648 (-2^{31}) в минимальное отрицательное значение типа `int`.

Если операнд данной операции имеет тип `ulong`, выводится сообщение об ошибке. Единственным исключением из этого правила является возможность преобразования десятичного целочисленного литерала –9223372036854775808 (–263) в минимальное отрицательное значение типа `long`.

Операция логического отрицания

Унарная операция логического отрицания имеет следующий синтаксис: знак операции `!`, за которым следует операнд. Поскольку эта операция может быть перегружена, то выполняемые ею действия зависят от ее конкретной реализации для типа операнда. Однако во всех случаях при реализации этой операции операнд преобразуется к типу аргумента выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

Для данной операции существует только одна предопределенная реализация:

```
bool   operator !(bool x);
```

Она осуществляет логическое отрицание операнда: если операнд имел значение `true`, то операция возвращает значение `false`. И наоборот: если операнд имел значение `false`, то операция возвращает значение `true`.

Операция побитового отрицания

Унарная операция логического отрицания имеет следующий синтаксис: знак операции `~`, за которым следует операнд. Поскольку эта операция может быть перегружена, то выполняемые ею действия зависят от ее конкретной реализации для типа операнда. Однако во всех случаях при реализации этой операции операнд преобразуется к типу аргумента выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

Ниже приведен список предопределенных операций побитового отрицания:

```
int    operator ~(int x);
uint   operator ~(uint x);
```

```
long   operator ~(long x);
ulong  operator ~(ulong x);
```

В результате выполнения этой операции каждый бит операнда меняет свое значение на противоположное.

Операция побитового отрицания определена и для перечислимых типов. При этом аргумент преобразуется к базовому типу, над полученным значением производится операция побитового отрицания, и ее результат преобразуется к перечислимому типу.

Префиксное инкрементирование и декрементирование

Унарная операция префиксного инкрементирования и декрементирования имеет следующий синтаксис: знак операции `++` или `--`, за которым следует операнд.

В качестве операнда операций префиксного инкрементирования и декрементирования могут выступать переменная или процедура доступа к свойству или индексатору. Результатом операции является значение, имеющее тип операнда.

Если в качестве операнда выступает процедура доступа к свойству или индексатору, то соответствующее свойство или индексатор должны иметь процедуры доступа как для чтения, так и для записи. Если в свойстве или индексаторе отсутствует процедура доступа `get` или `set`, компилятор выдает сообщение об ошибке.

Типы `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` и все перечислимые типы содержат предопределенные операции инкрементирования и декрементирования. Предопределенная операция инкрементирования добавляет `1` к своему аргументу, а предопределенная операция инкрементирования вычитает `1` из своего аргумента. В остальных случаях действия, выполняемые операциями инкрементирования и декрементирования, зависят от их реализации в конкретном типе.

В процессе выполнения приложения операции префиксного инкрементирования и декрементирования исполняются следующим образом:

- Если операнд имеет тип значения:
  - вычисляется значение выражения операнда для получения доступа к переменной;
  - значение этой переменной сохраняется;
  - для сохраненной переменной вызывается соответствующая операция;
  - новое значение сохраненной переменной записывается в область памяти, выделенную для данной переменной;

- значением выражения постфиксного инкремента или декремента является результат выполненной операции.

❒ Если в качестве операнда выступает процедура доступа:

- если процедура доступа не является статической, определяется объект, для которого вызывается данная процедура доступа;
- если вызывается процедура доступа к индексатору, вычисляются значения выражений в его списке выражений;
- вызывается соответствующая процедура доступа `get`, и во временной локальной переменной сохраняется возвращенное ей значение;
- для временной локальной переменной вызывается соответствующая операция;
- вызывается соответствующая процедура доступа `set`, которой в качестве аргумента `value` передается новое значение временной локальной переменной;
- значением выражения постфиксного инкремента или декремента является результат выполненной операции.

Предопределенные операции инкрементирования и декрементирования могут вызываться как в префиксной, так и в постфиксной формах. Результат операций инкрементирования и декрементирования в постфиксной форме соответствует значению операнда **до** выполнения операции, а результат этих операций в префиксной форме соответствует значению операнда **после** выполнения операции.

Перегруженные операции инкрементирования и декрементирования также могут вызываться в префиксной и в постфиксной формах, но при этом отсутствует возможность задать различные реализации операции для каждой из этих форм. Поэтому результат перегруженной операции инкрементирования или декрементирования не зависит от того, в какой форме она была вызвана.

Операция преобразования типа

Операция преобразования типа имеет следующий синтаксис: идентификатор типа, к которому производится преобразование, помещенный в круглые скобки, за которым следует унарное выражение. Если не существует явного преобразования из типа унарного выражения в тип, определяемый идентификатором типа, заключенным в круглые скобки, выдается сообщение об ошибке. Если же это преобразование существует, то результатом данного выражения является значение, имеющее указанный тип. Результатом данного выражения всегда является значение, даже в том случае, если его аргументом является переменная.

Поскольку один и тот же идентификатор может определять тип или переменную, то для определения того, является ли это выражение операцией преобразования типа или операцией в скобках, считается, что выражение в скобках является частью операции преобразования типов только в том случае, если выполняется хотя бы одно из следующих правил:

- последовательность идентификаторов в скобках имеет грамматику типа, неприменимую к выражению;
- последовательность идентификаторов в скобках имеет грамматику типа, а за закрывающей скобкой операции преобразования типа стоит символ `~`, символ `!`, символ `(`, идентификатор, литерал или любое ключевое слово, за исключением ключевого слова `is`.

Во всех остальных случаях принимается решение о том, что в скобках помещено выражение.

Применение таких сложных правил в такой, казалось бы, простой ситуации связано с тем, что при анализе содержимого скобок рассматривается только грамматика и не принимается в расчет значение используемых идентификаторов. Так, например, конструкция `Идентификатор.Идентификатор` может быть использована как для определения типа в пространстве имен, так и для доступа к члену класса.

Таким образом, применение этих правил позволяет определить, что конструкции `(First)Second`, `(First)(Second)` и `(First)(-Second)` используются для преобразования типа переменной `Second` к типу `First`, а конструкция `(First)-Second` представляет собой выражение для бинарной операции вычитания даже в том случае, если `First` является идентификатором типа. Единственным исключением из этого правила является преобразование к предопределенному типу, поскольку их идентификаторы зарезервированы и не могут использоваться в качестве идентификаторов переменных. Поэтому конструкция `(int)-Second` преобразует тип переменной `Second` к типу `int`.

Арифметические операции

К арифметическим операциям относят мультипликативные бинарные операции `*`, `/`, `%` и аддитивные бинарные операции `+` и `-`.

Операция умножения

Операция умножения является бинарной, поэтому знак операции `*` располагается между двумя ее операндами. Поскольку эта операция может быть перегружена, то выполняемые ею действия зависят от ее конкретной реализации для данных типов операндов. Однако во всех случаях при реализации этой операции оба операнда преобразуются к типу аргументов выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

Ниже приведен список предопределенных операций умножения:

```
int     operator *(int x, int y);
uint    operator *(uint x, uint y);
long    operator *(long x, long y);
ulong   operator *(ulong x, ulong y);
float   operator *(float x, float y);
double  operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

Если операнды операции умножения имеют тип `int`, `uint`, `long` или `ulong` и эта операция вызывается в операторе или операции `checked`, то при возникновении переполнения вызывается исключение `OverflowException`. Если это происходит в операторе или операции `unchecked`, то никакие сообщения об ошибках не выдаются, а в полученном результате отбрасываются все старшие биты, которые не могут быть сохранены в его формате.

Если операнды операции умножения имеют тип `float` или `double` и результат этой операции превышает по абсолютной величине диапазон возможных значений соответствующего типа, то результату присваивается значение "бесконечность" (выводится как `Infinity`) с сохранением его знака. При умножении этого значения на любое другое значение, кроме нулевого, результат также имеет значение "бесконечность". При умножении этого значения на нулевое значение результат не определен (выводится как `NaN`). При умножении неопределенного значения на любое другое значение результат также не определен. Если результат умножения двух ненулевых операндов оказывается меньше по абсолютной величине, чем наименьшее положительное значение данного типа, то результату присваивается нулевое значение с сохранением знака. Таким образом, в типах `float` или `double` нулевое значение имеет знак. Такие правила умножения определены в рекомендации `IEEE 754`.

Если операнды операции умножения имеют тип `decimal` и ее результат выходит за границы диапазона значений данного формата, то вызывается исключение `OverflowException`. Если результат операции слишком мал для представления в этом формате, для него устанавливается нулевое значение.

Операция деления

Операция деления является бинарной, поэтому знак операции `/` располагается между двумя ее операндами. Поскольку эта операция может быть перегружена, то выполняемые ею действия зависят от ее конкретной реализации для данных типов операндов. Однако во всех случаях при реализации этой операции оба операнда преобразуются к типу аргументов выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

Ниже приведен список предопределенных операций деления:

```
int     operator /(int x, int y);
uint    operator /(uint x, uint y);
long    operator /(long x, long y);
ulong   operator /(ulong x, ulong y);
float   operator /(float x, float y);
double  operator /(double x, double y);
decimal operator /(decimal x, decimal y);
```

Если операнды операции деления имеют тип `int`, `uint`, `long` или `ulong` и делитель имеет нулевое значение, то в любом случае вызывается исключение `DivideByZeroException`. Если делитель имеет ненулевое значение, то абсолютное результата данной операции представляет собой наибольшее целое, меньшее частного от деления операндов. Результат операции положителен, если оба операнда имеют одинаковые знаки, и отрицателен, если они имеют различные знаки.

Если операнды операции деления имеют тип `int` или `long`, делимое имеет максимальное по абсолютной величине отрицательное значение, а делитель равен `-1`, возникает переполнение. Если эта операция производилась в операции или операторе `checked`, вызывается исключение `OverflowException`. Если же вызов этой операции производился в операции или операторе `unchecked`, то результатом данной операции является значение делимого.

Если операнды операции деления имеют тип `float` или `double` и результат этой операции превышает по абсолютной величине диапазон возможных значений соответствующего типа, то результату присваивается значение "бесконечность" с сохранением его знака. Результат деления любого другого значения на значение "бесконечность" имеет нулевое значение. Результат деления бесконечности самой на себя или деления одного нулевого значения на другое не определен. Если хотя бы один из операндов операции деления имеет неопределенное значение, то и результат этой операции не определен. Такие правила деления определены в рекомендации `IEEE 754`.

Если операнды операции деления имеют тип `decimal` и делитель имеет нулевое значение, то вызывается исключение `DivideByZeroException`. Если результат операции слишком мал для представления в этом формате, для него устанавливается нулевое значение.

Операция деления по модулю

Операция деления по модулю является бинарной, поэтому знак операции `%` располагается между двумя ее операндами. Поскольку эта операция может быть перегружена, то выполняемые ею действия зависят от ее конкретной реализации для данных типов операндов. Однако во всех случаях при реали-

зации этой операции оба операнда преобразуются к типу аргументов выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

Ниже приведен список предопределенных операций деления по модулю:

```
int    operator %(int x, int y);
uint   operator %(uint x, uint y);
long   operator %(long x, long y);
ulong  operator %(ulong x, ulong y);
float  operator %(float x, float y);
double operator %(double x, double y);
decimaloperator %(decimal x, decimal y);
```

Результат операции деления по модулю представляет собой остаток от целочисленного деления операндов.

Если операнды операции деления по модулю имеют тип `int`, `uint`, `long` или `ulong` и делитель имеет нулевое значение, то вызывается исключение `DivideByZeroException`.

Если операнды операции деления по модулю имеют тип `float` или `double`, то результатом данной операции является неопределенное значение, если это значение имеет любой из ее операндов, если делимое имеет значение "бесконечность" или делитель имеет нулевое значение. Если делитель имеет значение "бесконечность", то результат данной операции совпадает с делителем независимо от знака делителя. В данном случае остаток от деления определяется как разность делителя и произведения наибольшего целого числа, не превышающего частное, на делитель. Это правило отличается от требований рекомендации `IEEE 754`, в которой остаток от деления определяется как разность делителя и произведения ближайшего к частному целого числа на делитель.

Если операнды операции деления имеют тип `decimal` и делитель имеет нулевое значение, то вызывается исключение `DivideByZeroException`. Если результат слишком велик для представления в этом формате, вызывается исключение `OverflowException`. Если результат операции слишком мал для представления в этом формате, для него устанавливается нулевое значение.

Операция сложения

Операция сложения является бинарной, поэтому знак операции `+` располагается между двумя ее операндами. Поскольку эта операция может быть перегружена, то выполняемые ею действия зависят от ее конкретной реализации для данных типов операндов. Однако во всех случаях при реализации этой операции оба операнда преобразуются к типу аргументов выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

Ниже приведен список предопределенных операций сложения:

```
int    operator +(int x, int y);
uint   operator +(uint x, uint y);
long   operator +(long x, long y);
ulong  operator +(ulong x, ulong y);
float  operator +(float x, float y);
double operator +(double x, double y);
```

Если операнды операции сложения имеют тип `int`, `uint`, `long` или `ulong` и эта операция вызывается в операторе или операции `checked`, то при возникновении переполнения вызывается исключение `OverflowException`. Если это происходит в операторе или операции `unchecked`, то никакие сообщения об ошибках не выдаются, а в полученном результате отбрасываются все старшие биты, которые не могут быть сохранены в его формате.

Если операнды операции сложения имеют тип `float` или `double`, то результатом данной операции является неопределенное значение, если это значение имеет любой из ее операндов, или если ее операнды имеют значение "бесконечность" и разные знаки. Во всех остальных случаях результат операции сложения предсказуем. Такие правила сложения определены в рекомендации `IEEE 754`.

Если операнды операции сложения имеют тип `decimal` и результат операции слишком велик для представления в этом формате, вызывается исключение `OverflowException`. Если результат операции слишком мал для представления в этом формате, для него устанавливается нулевое значение.

Кроме описанных выше предопределенных реализаций операции сложения, операция `+` стандартным образом перегружена в перечислимых типах, в типе `string` и в типах делегатов.

Если перечислимый тип `Enum_Type` имеет базовый тип `Underlying_Type` (представляющий собой один из целочисленных типов), то этот тип имеет две перегруженные операции

```
Enum_Type     operator +( Enum_Type x, Underlying_Type y);
Enum_Type     operator +( Underlying_Type x, Enum_Type y);
```

Обе эти операции реализуются как

```
(Enum_Type)( (Underlying_Type)x + (Underlying_Type)y )
```

Другими словами, данная операция увеличивает порядковый номер операнда, имеющего перечислимый тип, и возвращает значение перечислимого типа, соответствующее новому порядковому номеру.

В типе `string` определены следующие перегруженные операции `+`:

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

Эта операция производит конкатенацию строк, если хотя бы один из его аргументов имеет тип `string`. Если один из операндов данной операции имеет значение `null`, то к другому операнду добавляется пустая строка. В противном случае, если операнд не является строкой, то для него вызывается виртуальный метод `ToString()` унаследованный им от базового класса `object`. Если метод `ToString()` возвращает значение `null`, то к другому операнду добавляется пустая строка.

Результатом операции конкатенации строк является новая строка, состоящая из всех символов левого операнда, за которыми следуют все символы правого операнда. Если для размещения создаваемой строки недостаточно свободной оперативной памяти, вызывается исключение `OutOfMemoryException`.

Каждый тип делегата `Delegate_Type` содержит следующую реализацию операции `+`, осуществляющую конкатенацию делегатов:

```
Delegate_Type operator +( Delegate_Type x, Delegate_Type y);
```

Операция вычитания

Операция вычитания является бинарной, поэтому знак операции – располагается между двумя ее операндами. Поскольку эта операция может быть перегружена, то выполняемые ею действия зависят от ее конкретной реализации для данных типов операндов. Однако во всех случаях при реализации этой операции оба операнда преобразуются к типу аргументов выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

Ниже приведен список предопределенных операций вычитания:

```
int    operator -(int x, int y);
uint   operator -(uint x, uint y);
long   operator -(long x, long y);
ulong  operator -(ulong x, ulong y);
float  operator -(float x, float y);
double operator -(double x, double y);
```

Если операнды операции вычитания имеют тип `int`, `uint`, `long` или `ulong` и эта операция вызывается в операторе или операции `checked`, то при возникновении переполнения вызывается исключение `OverflowException`. Если это происходит в операторе или операции `unchecked`, то никакие сообщения об ошибках не выдаются, а в полученном результате отбрасываются все старшие биты, которые не могут быть сохранены в его формате.

Если операнды операции вычитания имеют тип `float` или `double`, то результатом данной операции является неопределенное значение, если это значение имеет любой из ее операндов, или если ее операнды имеют значение "бесконечность" и одинаковые знаки. Во всех остальных случаях результат операции вычитания предсказуем. Такие правила сложения определены в рекомендации `IEEE 754`.

Если операнды операции вычитания имеют тип `decimal` и результат операции слишком велик для представления в этом формате, вызывается исключение `OverflowException`. Если результат операции слишком мал для представления в этом формате, для него устанавливается нулевое значение.

Кроме описанных выше предопределенных реализаций операции вычитания, операция – стандартным образом перегружена в перечислимых типах и в типах делегатов.

Если перечислимый тип `Enum_Type` имеет базовый тип `Underlying_Type` (представляющий собой один из целочисленных типов), то этот тип имеет две перегруженные операции

```
Underlying_Type              operator -( Enum_Type x, Enum_Type y);
Enum_Type             operator -( Enum_Type x, Underlying_Type y);
```

Первая из этих операций реализуется как

```
(Underlying_Type)x — (Underlying_Type)y
```

Другими словами, данная операция вычисляет разность между позициями двух элементов перечислимого типа.

Вторая операция реализуется как

```
(Enum_Type)( (Underlying_Type)x — y )
```

Другими словами, данная операция уменьшает порядковый номер первого операнда и возвращает значение перечислимого типа, соответствующее новому порядковому номеру.

Каждый тип делегата `Delegate_Type` содержит следующую реализацию операции –, осуществляющую удаление делегата:

```
Delegate_Type operator -( Delegate_Type x, Delegate_Type y);
```

Операции сдвига

К *операциям сдвига* относятся операция сдвига влево (<<) и операция сдвига вправо (>>). Поскольку эти операции являются бинарными, знак операции располагается между двумя ее операндами. Поскольку эти операции могут быть перегружены, то выполняемые ими действия зависят от их конкретной реализации для данных типов операндов. Однако во всех случаях при реализации этой операции оба операнда преобразуются к типу аргументов выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

При перегрузке этих операций тип первого операнда обязательно должен совпадать с типом класса или структуры, в котором производится перегрузка этой операции, а второй операнд всегда должен иметь тип `int`.

Ниже приведен список предопределенных операций сдвига:

- Сдвиг влево

```
int    operator <<(int x, int count);
uint   operator <<(uint x, int count);
long   operator <<(long x, int count);
ulong  operator <<(ulong x, int count);
```

- Сдвиг вправо

```
int    operator >>(int x, int count);
uint   operator >>(uint x, int count);
long   operator >>(long x, int count);
ulong  operator >>(ulong x, int count);
```

Эти операции производят сдвиг битов первого операнда влево или вправо. При сдвиге влево старшие биты отбрасываются, а младшие биты заполняются нулями. При сдвиге вправо младшие биты всегда отбрасываются, а заполнение старших битов зависит от типа первого аргумента. Если первый аргумент имеет тип `int` или `long`, то старшие биты заполняются значением знакового разряда. Такой сдвиг называется *арифметическим*. Если же первый аргумент имеет тип `uint` или `ulong`, то старшие биты всегда заполняются нулевыми значениями. Такой сдвиг называется *логическим*.

В предопределенных операциях сдвига число бит, на которые должен осуществляться сдвиг, определяется следующим образом:

- Если первый операнд имеет значение `int` или `uint`, величина сдвига определяется пятью младшими битами второго операнда. Другими словами, можно считать, что перед выполнением данной операции вычисляется выражение `count &= 0x1F`.
- Если первый операнд имеет значение `long` или `ulong`, величина сдвига определяется шестью младшими битами второго операнда. Другими словами, можно считать, что перед выполнением данной операции вычисляется выражение `count &= 0x3F`.

Если соответствующие младшие биты второго операнда содержат нулевое значение, то результатом операции является значение первого операнда.

Операции сдвига никогда не вызывают переполнения и работают одинаково в операциях и операторах `checked` и `unchecked`.

Операции отношения

К *операциям отношения* относятся операции `==`, `!=`, `<`, `>`, `<=`, `>=` и `is`. Поскольку эти операции являются бинарными, знак операции располагается между двумя ее операндами. Все эти операции, кроме операции `is`, могут быть перегружены, поэтому выполняемые ими действия зависят от их кон-

кретной реализации для данных типов операндов. Однако во всех случаях при реализации этой операции оба операнда преобразуются к типу аргументов выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

Операции `==`, `!=`, `<`, `>`, `<=` и `>=` носят общее название операций сравнения. В следующих разделах будут описаны их предопределенные реализации для различных типов, однако, действия, выполняемые этими операциями, практически не зависят от типа их аргументов. Все они возвращают значение типа `bool`. Краткое описание выполняемых ими действий приведено в табл. 8.3. Во всех случаях, когда соответствующая операция не возвращает значение `true`, она возвращает значение `false`.

Таблица 8.3. *Операции сравнения*

Операция	Возвращает значение true, если
==	Первый операнд равен второму операнду
!=	Первый операнд не равен второму операнду
<	Первый операнд меньше второго операнда
>	Первый операнд больше второго операнда
<=	Первый операнд меньше либо равен второму операнду
>=	Первый операнд больше либо равен второму операнду

Целочисленные операции сравнения

Ниже приведен список предопределенных операций сравнения для целочисленных типов:

```
bool   operator ==(int x, int y);
bool   operator ==(uint x, uint y);
bool   operator ==(long x, long y);
bool   operator ==(ulong x, ulong y);
bool   operator !=(int x, int y);
bool   operator !=(uint x, uint y);
bool   operator !=(long x, long y);
bool   operator !=(ulong x, ulong y);
bool   operator <(int x, int y);
bool   operator <(uint x, uint y);
bool   operator <(long x, long y);
bool   operator <(ulong x, ulong y);
bool   operator >(int x, int y);
```

```
bool   operator >(uint x, uint y);
bool   operator >(long x, long y);
bool   operator >(ulong x, ulong y);
bool   operator <=(int x, int y);
bool   operator <=(uint x, uint y);
bool   operator <=(long x, long y);
bool   operator <=(ulong x, ulong y);
bool   operator >=(int x, int y);
bool   operator >=(uint x, uint y);
bool   operator >=(long x, long y);
bool   operator >=(ulong x, ulong y);
```

Значения, возвращаемые этими операциями, определены в табл. 8.3.

Операции сравнения для чисел с плавающей запятой

Ниже приведен список предопределенных операций сравнения для типов чисел с плавающей запятой:

```
bool   operator ==(float x, float y);
bool   operator ==(double x, double y);
bool   operator !=(float x, float y);
bool   operator !=(double x, double y);
bool   operator <(float x, float y);
bool   operator <(double x, double y);
bool   operator >(float x, float y);
bool   operator >(double x, double y);
bool   operator <=(float x, float y);
bool   operator <=(double x, double y);
bool   operator >=(float x, float y);
bool   operator >=(double x, double y);
```

Результаты этих операций определены в соответствии с рекомендацией IEEE 754:

- Если любой из операндов имеет неопределенное значение, то все операции, кроме операции `!=`, возвращают значение `false`, а операция `!=` возвращает значение `true`. Такое определение операций гарантирует, что при любых значениях `x` и `y` выражение `!(x == y)` будет иметь то же значение, что и выражение `x != y`. Это утверждение неверно в отношении остальных операций, поскольку, если один из операндов имеет неопределенное значение, выражения `!(x > y)` и `x <= y` будут иметь разные значения.

- В остальных случаях результат операции определяется в соответствии с табл. 8.3. При этом установлено следующее отношение величин, где `min` обозначает минимальное положительное значение данного типа, а `max` — максимальное положительное значение данного типа.

  $-\infty$ < -max < ... < -min < -0.0 == +0.0 < min < ... < max < $+\infty$

При рассмотрении этой последовательности следует обратить свое внимание на то, что:

- положительное нулевое значение равно отрицательному нулевому значению;
- величина "бесконечность" рассматривается как величина, большая максимального положительного значения данного типа, но равная другой величине "бесконечность".

Операции сравнения для десятичных чисел

Ниже приведен список предопределенных операций сравнения для десятичных чисел:

```
bool   operator ==(decimal x, decimal y);
bool   operator !=(decimal x, decimal y);
bool   operator <(decimal x, decimal y);
bool   operator >(decimal x, decimal y);
bool   operator <=(decimal x, decimal y);
bool   operator >=(decimal x, decimal y);
```

Значения, возвращаемые этими операциями, определены в табл. 8.3.

Логические операции сравнения

В логическом типе `bool` имеются предопределенные операции сравнения:

```
bool   operator ==(bool x, bool y);
bool   operator !=(bool x, bool y);
```

Как и в любом другом случае, операция `==` возвращает значение `true`, если оба операнда имеют одно и то же значение (оба имеют значение `true` или оба имеют значение `false`), и значение `false` в противном случае. Эта операция соответствует логической операции эквивалентности.

Операция `!=` возвращает значение `true`, если ее операнды имеют различные значения (один из них имеет значение `true`, а другой — значение `false`), и значение `false` в противном случае. Эта операция соответствует логической операции ИСКЛЮЧАЮЩЕГО ИЛИ.

Операции сравнения для перечислимых типов

В каждом перечислимом типе `Enum_Type` предопределены следующие операции сравнения:

```
bool   operator ==(Enum_Type x, Enum_Type y);
bool   operator !=(Enum_Type x, Enum_Type y);
bool   operator <(Enum_Type x, Enum_Type y);
bool   operator >(Enum_Type x, Enum_Type y);
bool   operator <=(Enum_Type x, Enum_Type y);
bool   operator >=(Enum_Type x, Enum_Type y);
```

При выполнении этих операций определяются порядковые номера, соответствующие операндам в данном типе, и сравнения производятся для этих порядковых номеров.

Операции сравнения для типов ссылок

При сравнении операндов, имеющих тип ссылки, используются следующие предопределенные операции класса `object`:

```
bool   operator ==(object x, object y);
bool   operator !=(object x, object y);
```

Поскольку в качестве операндов данных операций используются объекты класса `object`, то эти операции будут вызываться в любом случае, когда операнды операций `==` и `!=` не могут быть приведены к другому типу, перегружающему данные операции. С другой стороны, любая перегрузка операций `==` и `!=` скрывает данные предопределенные операции класса `object`.

Эти операции требуют, чтобы их операнды имели бы тип ссылки или значение `null`. Кроме того, требуется наличие двустороннего неявного преобразования из типа одного операнда в тип другого операнда. Если не выполнено хотя бы одно из поставленных выше условий, компилятор выдает сообщение об ошибке. Особенно важно запомнить следующие выводы из этих правил:

- Нельзя производить сравнение двух ссылок, которые не могут ссылаться на один и тот же объект. Например, нельзя сравнивать ссылки, имеющие типы классов, принадлежащих разным ветвям иерархического дерева. То есть в том случае, если один из классов не является потомком другого. При обнаружении попытки такого сравнения компилятор выдаст сообщение об ошибке.
- Нельзя сравнивать операнды, имеющие тип значения. Поэтому, если в структуре не перегружены соответствующие операции, нельзя сравнивать значения, имеющие тип данной структуры.

Переменные типов значений не преобразуются к типу `object`, поскольку эта операция не имеет смысла. Это связано с тем, что в результате этой операции получается ссылка на объект, соответствующий данной переменной, а на этот объект не может быть другой ссылки.

Если возникает необходимость проверить эквивалентность ссылок на объект, в типе которого перегружены операции `==` и `!=`, эти ссылки должны быть явным образом преобразованы к типу `object`. Рассмотрим это на примере:

```
class Demo
{
        static void Main()
        {
                string First  = "Some string";
                string Second = string.Copy(First);
                if(First == Second)
                        Console.WriteLine("OK");
                if((object) First != (object)Second)
                        Console.WriteLine("OK");
        }
}
```

В результате выполнения этой программы выводятся две строки **OK**, свидетельствующие о том, что хотя переменные `First` и `Second` содержат одну и ту же строку, они ссылаются на различные объекты.

Операция сравнения строк

Тип `string` содержит следующие предопределенные операции сравнения:

```
bool   operator ==(string x, string y);
bool   operator !=(string x, string y);
```

Два выражения, имеющие тип `string`, считаются равными, если:

- оба они имеют значение `null`;
- оба они ссылаются на объекты строк, имеющих одинаковую длину и совпадающие символы в каждой своей позиции.

Эта операция сравнивает значения строк, а не ссылки на них. Поэтому в примере, приведенном в предыдущем разделе, две строки, содержащиеся в различных объектах, оказываются равными.

Операция сравнения делегатов

В каждом типе делегата `Delegate_Type` предопределены следующие операции сравнения:

```
bool   operator ==(Delegate_Type x, Delegate_Type y);
bool   operator !=(Delegate_Type x, Delegate_Type y);
```

Операция is

Операция `is` используется для проверки совместимости типов в процессе выполнения приложения. Данная операция имеет следующий синтаксис: выражение, имеющее тип ссылки, за которым следует ключевое слово `is` и идентификатор типа ссылки. Эта операция не может вызвать ошибки в процессе компиляции.

Операция `is` возвращает значение `true`, если выражение, имеющее тип ссылки, на момент проверки имеет ненулевое значение и тип, имеющий неявное преобразование к типу, определяемому идентификатором типа ссылки. Для простоты можно считать, что данная операция проверяет, имеет ли выражение ненулевое значение, и производит неявное преобразование к требуемому типу. Если в процессе преобразования не было вызвано исключения, эта операция возвращает значение `true`. В противном случае она возвращает значение `false`.

Если тип выражения известен уже на момент компиляции приложения, то есть, если эта операция всегда возвращает значение `true` или всегда возвращает значение `false`, то компилятор выдает сообщение об ошибке.

Чаще всего операция `is` используется для проверки того, поддерживает ли объект данный интерфейс.

Операция as

Операция `as` во многом аналогична операции `is`, но она не только позволяет выяснить возможность преобразования типа ссылки на объект, но и произвести преобразование этой ссылки к требуемому типу. Если преобразование невозможно, результатом операции `as` является значение `null`.

Побитовые логические операции

К побитовым логическим операциям относятся операции `&`, `^` и `|`. Поскольку эти операции являются бинарными, знак операции располагается между двумя ее операндами. Все эти операции, кроме операции `is`, могут быть перегружены, поэтому выполняемые ими действия зависят от их конкретной реализации для данных типов операндов. Однако во всех случаях при реали-

зации этой операции оба операнда преобразуются к типу аргументов выбранной операции, а ее результат имеет тот тип, в котором была перегружена данная операция.

В следующих разделах будут рассмотрены предопределенные побитовые логические операции для различных типов данных.

Целочисленные побитовые логические операции

Ниже приведен список предопределенных побитовых логических операций для целочисленных типов:

```
int    operator &(int x, int y);
uint   operator &(uint x, uint y);
long   operator &(long x, long y);
ulong  operator &(ulong x, ulong y);
int    operator |(int x, int y);
uint   operator |(uint x, uint y);
long   operator |(long x, long y);
ulong  operator |(ulong x, ulong y);
int    operator ^(int x, int y);
uint   operator ^(uint x, uint y);
long   operator ^(long x, long y);
ulong  operator ^(ulong x, ulong y);
```

Предопределенная операция `&` производит побитовую операцию логического И над своими операндами, предопределенная операция `|` производит над ними побитовую операцию логического ИЛИ, а предопределенная операция `^` — операцию ИСКЛЮЧАЮЩЕГО ИЛИ. Эти операции не могут вызвать исключения.

Побитовые логические операции для перечислимых типов

В каждом перечислимом типе `Enum_Type` предопределены следующие побитовые логические операции:

```
Enum_Type operator &(Enum_Type x, Enum_Type y);
Enum_Type operator |(Enum_Type x, Enum_Type y);
Enum_Type operator ^(Enum_Type x, Enum_Type y);
```

Побитовые логические операции производятся над порядковыми номерами операндов в типе, а результат этой операции используется как порядковый номер результата в типе.

Побитовые логические операции для логического типа

В логическом типе `bool` имеются предопределенные побитовые логические операции:

```
bool   operator &(bool x, bool y);
bool   operator |(bool x, bool y);
bool   operator ^(bool x, bool y);
```

Поскольку предполагается, что логическое значение состоит всего из одного бита, то возвращаемое значение предопределенной операции совпадает с результатом соответствующей логической операции. Результаты этих операций приведены в табл. 8.4.

Таблица 8.4. *Логические операции*

x	y	Операция		
		&	|	^
false	false	false	false	false
false	true	false	true	true
true	false	false	true	true
true	true	true	true	false

Логические операции

К логическим операциям относятся операции `&&` и `||`. Эти операции могут производиться только над выражениями логического типа. Операция `&&` соответствует операции `&`, произведенной над выражением логического типа, а операция `||` соответствует операции `|`, произведенной над выражениями того же типа. Основным отличием данных операций от побитовых операций является то, что логические операции могут выполняться не полностью. Операция `&&` возвращает значение `false`, если ее первый операнд имеет значение `false`. При этом значение второго операнда не определяется. Аналогично операция `||` возвращает значение `true`, если ее первый операнд имеет значение `true`. При этом значение второго операнда не определяется.

Подобный способ выполнения операций позволяет помещать во второй операнд выражения, допустимость вычисления значений которых зависит от значения первого операнда. Например, в первом операнде операции `&&` может проверяться наличие объекта, а во втором — производится обращение к данному объекту.

Операции `&&` и `||` не могут быть перегружены, но поскольку они включают в себя выполнение побитовых логических операций, то их перегрузка одновременно означает и перегрузку соответствующих операций `&&` и `||`. Поэтому:

- При вызове логических операций производится поиск наилучшей реализации соответствующих побитовых логических операций. Если этот поиск не позволяет однозначно определить наилучшего кандидата на реализацию данной операции, или если в результате поиска выбирается одна из целочисленных предопределенных операций, то выдается сообщение об ошибке.
- Если в результате поиска была выбрана одна из предопределенных операций для логического типа, или в типе операндов отсутствуют перегруженные операции `&` или `|`, но определено неявное преобразование к типу `bool`:
  - операция `First && Second` выполняется как `First? Second: false`. То есть операнд `First` преобразуется к типу `bool` и производится оценка его значения. Если операнд `First` имеет значение `true`, производится преобразование значения операнда `Second` к типу `bool` и полученное значение рассматривается в качестве возвращаемого значения операции. В противном случае возвращается значение `false`;
  - операция `First || Second` выполняется как `First? true: Second`. То есть, операнд `First` преобразуется к типу `bool` и производится оценка его значения. Если операнд `First` имеет значение `true`, то операция возвращает значение `true`. В противном случае производится преобразование значения операнда `Second` к типу `bool` и полученное значение рассматривается в качестве возвращаемого значения операции.
- В противном случае производится поиск перегруженной операции `&` или `|` и ее выполнение по следующим правилам:
  - производится проверка того, что тип возвращаемого значения и типы операндов совпадают с типом `Some_Type`, в котором перегружена данная операция. В этом типе должны быть определены операции `true` и `false`. Если хотя бы одно из этих условий не выполняется, компилятор выдает сообщение об ошибке;
  - если все приведенные выше условия выполняются, то производится определение возвращаемых значений операции `&&` или `||` с использованием операций `Some_Type.true` и `Some_Type.false`;
  - операция `First && Second` выполняется как `Some_Type.false(First)? First: Some_Type.&(First,Second)`. То есть, для операнда `First` вызывается перегруженная операция `false`, проверяющая его логическое значение, если операнд `First` имеет значение `false`, то операция `&&` возвращает значение этого операнда. В противном случае определяется значение операнда `Second` и обоих операндов, и вызывается перегруженная операция `&`, результат которой и является результатом операции `&&`;

- операция `First || Second` выполняется как `Some_Type.true(First)? First: Some_Type.|(First,Second)`. То есть для операнда `First` вызывается перегруженная операция `true`, проверяющая его логическое значение, если операнд `First` имеет значение `true`, то операция `||` возвращает значение этого операнда. В противном случае определяется значение операнда `Second` и обоих операндов, и вызывается перегруженная операция `|`, результат которой и является результатом операции `||`.

Такие сложные преобразования при работе с перегруженными операциями связаны с тем, что побитовые логические операции возвращают тип своих операндов.

Условная операция

Условная операция является тернарной и имеет следующий синтаксис: выражение условия, знак операции `?`, выражение, выполняемое, если выражение условия имеет значение `true`, знак операции `:`, выражение, выполняемое в противном случае. В условной операции определяются значения только для условного выражения и одного из исполняемых выражений. Значение другого исполняемого выражения не определяется.

Первым операндом условной операции должно быть выражение, тип которого имеет неявное преобразование к типу `bool`, или выражение, в типе которого перегружена операция `true`. Если первый операнд не удовлетворяет ни одному из перечисленных выше условий, компилятор выдает сообщение об ошибке 0.

Типы второго и третьего операндов условной операции определяют тип возвращаемого значения этой операции. Тип возвращаемого значения условной операции определяется в процессе компиляции следующим образом:

- если второй и третий ее операнды имеют один и тот же тип, то тот же тип имеет и возвращаемое значение условной операции;
- если существует только одностороннее преобразование из типа одного операнда в тип другого операнда, то выходное значение условного оператора имеет тип того операнда, для которого отсутствует преобразование к типу другого операнда;
- если приведенные выше правила не могут быть применены, компилятор выдает сообщение об ошибке.

В процессе выполнения приложения условный оператор исполняется следующим образом:

- Вычисляется значение первого выражения и преобразуется к типу `bool`:
  - если существует неявное преобразование типа первого выражения в тип `bool`, то используется это неявное преобразование;

- если же такое преобразование отсутствует, вызывается перегруженная в типе первого выражения операция `true`, возвращающая значение типа `bool`.

- Если в результате выполнения операций на предыдущем шаге получено логическое значение `true`, то вычисляется второе выражение и его значение становится выходным значением условной операции. При необходимости при этом производится преобразование типа данного выражения к типу возвращаемого значения условной операции.
- В противном случае вычисляется третье выражение в и его значение становится выходным значением условной операции. При необходимости при этом производится преобразование типа данного выражения к типу возвращаемого значения условной операции.

Операции присваивания

Операции присваивания используются для присваивания новых значений переменным, свойствам или элементам индексатора. К операциям присваивания относятся операции `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=` и `>>=`.

Слева от знака операции присваивания может располагаться переменная или процедура доступа к свойству или индексатору.

Помимо простой операции присваивания (`=`), производящей только присваивание нового значения, определены также составные операции присваивания, в которых перед знаком простой операции присваивания помещается знак одной из бинарных операций. Составные операции присваивания можно рассматривать как последовательное выполнение двух операций: сначала выполняется бинарная операция, знак которой стоит перед знаком простой операции присваивания (операция выполняется так, как если бы в составной операции присваивания отсутствовал бы знак простой операции присваивания), а затем результат этой операции подставляется на место второго операнда составной операции присваивания и выполняется простая операция присваивания.

Операции присваивания выполняются справа налево. Например, последовательность операций `x = y = z` эквивалентна последовательности операций `x = (y = z)`.

Простая операция присваивания

Как уже говорилось выше, операция `=` называется простой операцией присваивания. Правый операнд данной операции должен представлять собой выражение, тип которого имеет неявное преобразование к типу левого операнда. Результатом выполнения простой операции присваивания является

присваивание значения ее правого операнда переменной или процедуре доступа к свойству или идентификатору, выступающему в качестве левого операнда данной операции.

Если в качестве левого операнда выступает процедура доступа к свойству или индексатору, то данное свойство или индексатор должны иметь процедуру доступа `set`. В противном случае компилятор выдаст сообщение об ошибке.

В процессе выполнения приложения операция простого присваивания выполняется следующим образом:

- Если в качестве левого операнда данной операции выступает переменная:
  - получается доступ к данной переменной;
  - вычисляется значение правого операнда и при необходимости преобразуется к типу левого операнда;
  - если в качестве левого операнда данной операции выступает элемент массива, имеющий тип ссылки, в процессе выполнения приложения производится проверка того, что вычисленное значение правого операнда имеет тип, совместимый с объектом массива, элементом которого является левый операнд. Тип является совместимым, если правый операнд имеет значение `null` или существует неявное преобразование текущего типа объекта, на который ссылается правый операнд, к текущему типу объекта массива, которому принадлежит левый операнд. Если эта проверка не завершилась успешно, вызывается исключение `ArrayTypeMismatchException`;
  - значение правого операнда сохраняется в области памяти, выделенной для хранения значения левого операнда.
- Если в качестве левого операнда данной операции выступает процедура доступа к свойству или индексатору:
  - если данная процедура доступа не является статической, определяется объект, для которого она вызывается;
  - если данная процедура осуществляет доступ к индексатору, вычисляются значения выражений в ее списке аргументов;
  - вычисляется значение правого операнда и при необходимости преобразуется к типу левого операнда;
  - вызывается соответствующая процедура доступа `set`, которой в качестве аргумента `value` передается значение правого операнда.

Необходимость проверки в процессе выполнения приложения совместимости типов присваиваемого выражения и элемента массива связана с тем, что при наличии неявного преобразования из типа `First_Type` в тип `Second_Type`, существует также неявное преобразование из типа `First_Type[]` в тип

Second_Type[]. Поэтому элементы массива First_Type[] могут храниться в элементах массива Second_Type[], однако, при этом они продолжают оставаться объектами типа First_Type. Например:

```
string[] str = new string[8];
object[] obj = str;
obj[0] = null;
obj[1] = "Demo";
obj[2] = new int[10]; // Вызывается исключение ArrayTypeMismatchException
```

При попытке присваивания третьему элементу массива значения типа int[] вызывается исключение ArrayTypeMismatchException, поскольку этот тип не может быть присвоен объекту класса string.

Если присваивание производится процедуре доступа к свойству или индексатору, определенной в типе структуры, то для обращения к этой процедуре может использоваться только переменная. Если для этого будет использовано значение, компилятор выдаст сообщение об ошибке.

Составная операция присваивания

Как уже говорилось выше, составную операцию присваивания можно разбить на две операции: дополнительную бинарную операцию и простую операцию присваивания. Эти операции выполняются следующим образом:

- Производится поиск последней перегрузки дополнительной бинарной операции.
- Если тип возвращаемого значения выбранной операции имеет **неявное** преобразование к типу левого операнда, сначала выполняется бинарная операция, знак которой стоит перед знаком простой операции присваивания. Операция выполняется так, как если бы в составной операции присваивания отсутствовал бы знак простой операции присваивания. Результат этой операции подставляется на место второго операнда составной операции присваивания и выполняется простая операция присваивания.
- Если в процессе поиска последней перегрузки выбрана предопределенная операция и возвращаемый тип этой операции имеет **явное** преобразование к типу левого операнда, а правый операнд имеет **неявное** преобразование к типу левого операнда, то в простой операции присваивания используется явное преобразование типа результата дополнительной бинарной операции к типу левого операнда. Это правило используется для того, чтобы составные операции присваивания могли бы иметь аргументы типов sbyte, byte, short, ushort или char. Дело в том, что бинарные операции определены таким образом, что, даже если оба их операнда имеют один из перечисленных выше типов, то результат операции все равно имеет тип int.
- Если не могут быть применены описанные выше преобразования типов, компилятор выдает сообщение об ошибке.

Описанное выше разбиение составной операции присваивания на две отдельные операции не совсем корректно. Дело в том, что при реализации каждой операции производится вычисление адреса области памяти, в которой хранятся ее операнды. При описанном выше разбиении на две операции адрес левого операнда должен вычисляться дважды: отдельно для каждой операции. Однако, поскольку составная операция присваивания представляет собой одну операцию, адрес левого операнда вычисляется только один раз. После чего он сохраняется во временной переменной и используется для всех последующих действий с этим операндом.

Если в качестве левого операнда составной операции присваивания выступает процедура доступа к свойству или индексатору, соответствующее свойство или индексатор должны иметь как процедуру доступа `get`, так и процедуру доступа `set`. При отсутствии любой из этих процедур компилятор выдает сообщение об ошибке.

Правила выполнения составной операции присваивания составлены таким образом, чтобы она была запрещена в тех случаях, когда запрещена простая операция присваивания. Для того чтобы сделать операцию возможной, необходимо прибегнуть к явному преобразованию типов.

Константное выражение

Константное выражение представляет собой выражение, значение которого может быть определено в процессе компиляции. Константное выражение может иметь тип `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string` или любой перечислимый тип. Константное выражение может иметь значение `null`.

В константном выражении могут быть использованы:

- литералы (включая литерал `null`);
- ссылки на члены классов или структур, имеющие модификатор `const`;
- ссылки на члены перечислимых типов;
- константные выражения в скобках;
- преобразования типов для перечисленных выше константных выражений;
- предопределенные операции `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=` и `=>`, операндами которых являются перечисленные выше константные выражения;
- условная операция.

Если выражение состоит из перечисленных выше константных выражений, над которыми производятся перечисленные выше операции, то это выраже-

ние вычисляется в процессе компиляции. Это справедливо и в том случае, если данное константное выражение является частью другого выражения, не являющегося константным. При этом используются те же правила поиска наилучшей реализации, которые применяются в процессе выполнения приложения для выражений, не являющихся константными. Единственное отличие заключается в том, что вместо вызова исключения при вычислении константного выражения компилятор выдает сообщение об ошибке.

Если константное выражение не расположено в операции или операторе `unchecked`, при возникновении переполнения в процессе выполнения целочисленных операций или в процессе преобразования целочисленных типов при вычислении константных выражений компилятор выдает сообщение об ошибке.

Как уже говорилось выше, константные выражения могут являться составной частью обычных выражений. Однако существуют случаи, когда разрешено использование только константных выражений. Это требование распространяется на:

- объявление констант;
- объявление членов перечислимого типа;
- в метках `case` и `goto case` оператора-переключателя (`switch`);
- в атрибутах.

Поскольку всем целым числам с C# присваивается тип `int`, то для простоты константных выражений, определены неявные преобразования константных выражений из типа `int` к типам `sbyte`, `byte`, `short`, `ushort`, `uint` и `ulong`. Единственным требованием при этом является то, чтобы значение преобразуемого выражения находилось в пределах диапазона допустимых значений типа, к которому производится преобразование.

Логическое выражение

Логическим выражением называется выражение, возвращающее значение, имеющее тип `bool`. Логические выражения используются в качестве аргумента условного оператора и оператора цикла. Первое выражение условной операции (`?:`) подчиняется тем же правилам, что и логическое выражение, однако, исходя из требований, предъявляемых порядком выполнения операций, его рассматривают как логическую операцию.

Тип логического выражения должен иметь неявное преобразование к типу `bool` или к типу, в котором перегружена операция `true`. В последнем случае после вычисления значения данного выражения оно передается в качестве аргумента операции `true`, возвращающей значение типа `bool`.

Глава 9

Операторы

C# содержит обширный набор операторов. Большинство этих операторов унаследовано им от языков программирования C и C++. Различают операторы с метками, операторы объявлений и вложенные операторы.

К вложенным операторам относятся:

- блок;
- оператор-выражение;
- оператор цикла;
- оператор `try`;
- оператор `unchecked`;
- пустой оператор;
- оператор выбора;
- оператор перехода;
- оператор `checked`;
- оператор `lock`.

Конечная точка и доступность

Каждый оператор имеет *конечную точку*. Под конечной точкой можно понимать точку, расположенную за данным оператором. Если оператор является вложенным, то действия, предпринимаемые при достижении его конечной точки, определяются внешним оператором.

Если при каком-нибудь значении переменных приложения оператор может быть выполнен, он называется *доступным*. В противном случае он называется *недоступным*. Рассмотрим это на следующем примере:

```
void Some_Func()
{
        int     i;
        if( true )
                i = 0; // Доступный оператор
        else
                i = 1; // Недоступный оператор
}
```

Поскольку аргумент данного условного оператора имеет постоянное значение, одна из его ветвей становится недоступной. При появлении в приложении недоступных операторов компилятор выдает соответствующее предупреждение, поскольку наличие недоступных операторов ошибкой не является.

При определении доступности определенного оператора или конечной точки компилятор проверяет значения всех констант, которые могут повлиять на их доступность. При этом не проверяются значения, которыми инициализируются переменные. Это относится и к тем случаям, когда между инициализацией переменной и условным оператором, проверяющим ее значение, отсутствуют операторы, которые могут изменить значение данной переменной. Например, в приведенном ниже фрагменте программы все операторы считаются доступными:

```
void Some_Func()
{
        bool   b = true;
        int    i;
        if( b )
               i = 0; // Доступный оператор
        else
               i = 1; // Считается доступным
}
```

Для определения доступности операторов применяется следующий набор правил:

- блок функции всегда считается доступным, поэтому его первый оператор всегда доступен;
- если оператор доступен и этот оператор не является оператором перехода, то доступна и его конечная точка;
- оператор, расположенный за доступной конечной точкой является доступным;
- помеченный оператор является доступным, если его метка использована в операторе `goto`, который может передать управление этому помеченному оператору;
- вложенный оператор условного оператора является недоступным только в том случае, если аргумент условного оператора является константой, значение которой запрещает доступ в ту ветвь условного оператора, в которой расположен данный вложенный оператор.

Недоступными являются конечные точки следующих операторов:

- `return`;
- `break`;
- `goto`.

Конечная точка оператора не должна достигаться в двух случаях:

- в секции оператора-переключателя конечная точка последовательности операторов не должна быть доступной, поскольку в противном случае управление должно быть передано к следующей секции;
- конечная точка блока функции, возвращающей значения, должна быть недоступной, поскольку в этом случае не определено возвращаемое значение функции.

Блоки

Блок используется для объединения нескольких операторов в один составной оператор. Блок состоит из фигурных скобок, в которых может располагаться последовательность операторов.

Если в блоке содержатся операторы объявления, то область действия объявленных ими переменных или констант простирается от данного оператора объявления до конца блока. В блоке не могут быть объявлены два идентификатора, имеющих одно и то же имя, но различные типы или значения. Это относится не только к переменным, определенным в данном блоке, но и к наследуемым им переменным.

Блок исполняется следующим образом:

- если блок не содержит последовательности операторов, управление передается в конечную точку блока;
- в ином случае управление передается первому оператору в последовательности. Если управление передается конечной точке последовательности операторов, оно передается конечной точке блока.

Последовательность операторов доступна только в том случае, если доступен содержащий ее блок. Конец блока доступен, если он пуст или если доступна конечная точка последовательности операторов.

Последовательность операторов

Последовательность операторов представляет собой оператор или последовательность операторов, используемую в блоках и секциях операторов `switch`.

При передаче управления последовательности операторов управление передается ее первому оператору. При достижении конечной точки оператора управление передается следующему оператору. При достижении конечной точки последнего оператора списка управление передается конечной точке последовательности операторов.

Оператор в последовательности операторов доступен, если:

- он является первым оператором последовательности, и сама последовательность доступна;
- доступна конечная точка предыдущего оператора;
- оператор имеет метку, использованную в операторе `goto`, который может передать по ней управление.

Конечная точка последовательности операторов доступна, если доступна конечная точка последнего оператора последовательности.

Пустой оператор

Пустой оператор не выполняет никаких действий и состоит из точки с запятой.

Пустой оператор используется в том случае, если нет необходимости выполнять какую-либо операцию, но синтаксис требует присутствия оператора. В результате его выполнения управление просто передается его конечной точке. Поэтому конечная точка доступного пустого оператора всегда является доступной.

Пустой оператор используется, например, в операторе `while`, все обработки которого производятся в его аргументе:

```
bool Some_Func(int[] array, int length, int sample )
{
        int    i = 0;

        while ((array[i++] != sample) && ( i < length));

        if( i < length )
                return true;
        else
                return false;
}
```

Пустой оператор также часто используется в качестве последнего оператора последовательности, когда необходимо обеспечить доступность конечной точки последовательности операторов, а последний исполняемый оператор метить нельзя. Тогда за ним помещается помеченный пустой оператор.

Помеченный оператор

Помеченный оператор представляет собой оператор, перед которым поставлена метка. Метка представляет собой идентификатор, отделяемый от помечаемого оператора двоеточием. Можно пометить блок, но нельзя пометить вложенный оператор.

Область действия метки распространяется на блок, в котором она определена, включая все вложенные в него блоки. В одной области действия не могут быть определены две одинаковые метки. Однако метки не конфликтуют в своей области действия с другими идентификаторами, что позволяет определять поля и переменные с тем же именем, что и метки.

Метки используются для перехода к ним по оператору `goto`, расположенному в их области действия. То есть данный оператор может передавать управление в пределах одного блока или во внешний блок, но не может передавать управление во вложенный блок. Например:

```
bool Some_Func(int[] array, int length, int sample )
{
        for( int i = 0; i < length; i++)
        {
                if( array[i] == sample )
                        goto i;
        }

i:      if( i < length )
                return true;
        else
                return false;
}
```

Помеченный оператор является доступным, если эта метка использована в операторе `goto`, который может передать управление данному помеченному оператору.

Оператор объявления

Операторы объявления используются для объявления локальных переменных и констант. Как уже говорилось выше, оператор объявления не является вложенным оператором. Это связано с тем, что область действия локальной переменной распространяется от объявления переменной до конца блока, в котором она была определена. В данном случае это правило не может быть применено, поскольку область действия локальной переменной должна распространяться только на вложенный оператор, а не до конца блока, в котором использован внешний оператор.

В C++ объявление переменной во вложенном операторе не считается ошибкой. Транслятор просто выдает сообщение о том, что переменная была определена, но нигде не использовалась.

Объявление локальных переменных

Объявление локальных переменных состоит из типа и списка объявлений переменных. Объявления переменных в списке разделяются запятой. Объявление переменной состоит из идентификатора, за которым может следовать инициализатор, отделяемый от инициализатора знаком равенства. В зависимости от типа инициализируемой переменной инициализатор может представлять собой выражение, тип которого имеет неявное преобразование к типу переменной или инициализатор массива совместимого типа.

Локальная переменная должна быть инициализирована до первого использования своего значения. Область действия локальной переменной простирается от ее объявления до конца блока, в котором она определена. В пределах области действия локальной переменной не может быть определена другая переменная или константа с тем же именем.

Объявление переменных в одной строке полностью эквивалентно определению тех же самых переменных, каждой в своей строке, с указанием того же типа и инициализатора. Определенные до этого локальные переменные могут использоваться для инициализации других переменных. Таким образом, фрагмент программы

```
void Some_Func() {
        int    i = 3, j = 1 << i;
        int[]  a = new int[j];
}
```

эквивалентен фрагменту программы

```
void Some_Func() {
        int    i = 3;
        int    j = 1 << i;
        int[]  a = new int[j];
}
```

Все локальные объекты классов должны инициализироваться перед своим использованием. Поскольку все члены класса инициализируются при создании его объекта своими значениями по умолчанию, нет необходимости инициализировать их в конструкторе класса. В этом случае компилятор просто выдаст предупреждение о том, что будет использовано значение, назначенное члену класса по умолчанию. Ниже приведен пример, содержащий неинициализированный и инициализированный локальные объекты класса:

```
using System;

class  MyClass
{
```

```
        public MyClass( int i )
        {
                this.i = i;
        }

        public void Dump()
        {
                Console.WriteLine( "{0} {1}", i, j );
        }

        public int    i;
                int j;

}

class Test
{
        public static void Main()
        {
                MyClassFirst;
                MyClassSecond = new MyClass( 5 );

                First.Dump(); // Ошибка! Объект класса не инициализирован
                Second.Dump();
        }
}
```

При объявлении локальных структур действуют иные правила. Хотя при создании экземпляра структуры все ее поля инициализируются нулевыми значениями, отсутствие инициализации структуры рассматривается компилятором как ошибка. Однако в данном случае инициализация может производиться не только конструктором структуры, но и непосредственным присваиванием значений ее членам. Кроме того, в отличие от классов наличие в структуре конструктора не скрывает ее конструктор, используемый по умолчанию. Все это продемонстрировано в приведенном ниже примере:

```
using System;

struct MyStruct
{
        public MyStruct( int i )
        {
                this.i = i;
        }
```

```
        public void Dump()
        {
                Console.WriteLine( i );
        }

        public int    i;
}

class Test
{
        public static void Main()
        {
                MyStruct      First  = new MyStruct();
                MyStruct      Second = new MyStruct( 5 );
                MyStruct      Third;

                Third.i= 1;

                First.Dump();
                Second.Dump();
                Third.Dump();
                Console.ReadLine();
        }
}
```

Отдельно следует остановиться на инициализации локальных массивов. Дело в том, что доступ к элементам массивов, представляющих собой структуры или ссылки на объекты классов, возможен даже в том случае, если эти структуры или объекты классов не были инициализированы. Это естественно, поскольку в противном случае эти элементы не могли бы быть инициализированы. Поскольку проведение некоторых операций (например, вызов для них метода `Reverse`) с неинициализированными массивами может привести к возникновению трудноуловимых ошибок, а компилятор отказывается разбираться с подобными ситуациями, разработчику следует знать о них и обращать на них особое внимание.

Объявление локальных констант

Объявление локальных констант во многом аналогично объявлению локальных переменных с их одновременной инициализацией. Основными отличиями является то, что при объявлении констант перед типом помещается ключевое слово `const`, а в инициализаторе может использоваться только константное выра-

жение. Требования к типу константы и к константному выражению, используемому для его инициализации, полностью совпадают с требованиями, предъявляемыми к константам, являющимся членами классов.

Область действия локальной константы распространяется от ее объявления до конца блока, в котором она определена. В пределах области действия локальной переменной не может быть определена другая переменная или константа с тем же именем.

Оператор-выражение

Оператор-выражение представляет собой выражение, за которым следует точка с запятой. Полученное при этом значение не сохраняется. Это выражение может содержать:

- операцию вызова функции;
- выражение создания объекта;
- операцию присваивания;
- операцию постфиксного инкремента;
- операцию постфиксного декремента;
- операцию префиксного инкремента;
- операцию префиксного декремента.

C# выполняет операторы-выражения, вычисляя выражения. Используемое в операторе вычисления выражение обязательно должно иметь побочный эффект, который должен завершаться до начала следующего оператора. В операторе-выражении не могут, например, использоваться выражения типа `i*k` или `b == 3`, поскольку они просто вычисляют выражение, но не имеют побочных эффектов.

После вычисления выражения управление передается следующей конечной точке оператора.

Условный оператор

Условный оператор позволяет выбрать один из вложенных операторов в зависимости от значения своего аргумента, в качестве которого может выступать любое логическое выражение. Условный оператор состоит из ключевого слова `if`, за которым следуют круглые скобки, в которых размещается аргумент оператора и вложенный оператор, выполняемый в том случае, если аргумент условного оператора имеет значение `true`. За вложенным оператором может размещаться ключевое слово `else`, за которым следует вложенный оператор, выполняемый в том случае, если аргумент условного оператора имеет значение `false`.

Вложенный оператор, следующий за ключевым словом `else`, относится к ближайшему предшествующему оператору `if`, не содержащему ключевого слова `else`. Например, оператор

```
if ( bFirst )
        if ( bSecond )
                Some_Func();
        else
                Other_Finc();
```

эквивалентен оператору

```
if ( bFirst )
{
        if ( bSecond )
                Some_Func();
        else
                Other_Finc();
}
```

Условный оператор выполняется следующим образом:

- вычисляется значение логического выражения аргумента;
- если логическое выражение имеет значение `true`, управление передается первому вложенному оператору. Если управление передается конечной точке вложенного оператора, оно передается конечной точке условного оператора;
- если логическое выражение имеет значение `false` и в условном операторе отсутствует ключевое слово `else`, управление передается конечной точке условного оператора;
- если логическое выражение имеет значение `false` и в условном операторе присутствует ключевое слово `else`, управление передается вложенному оператору, следующему за ключевым словом `else`. Если управление передается конечной точке вложенного оператора, оно передается конечной точке условного оператора.

Первый вложенный оператор считается недоступным только в том случае, если аргумент условного оператора является константным выражением, и это выражение имеет значение `false`. Аналогично второй вложенный оператор считается недоступным только в том случае, если аргумент условного оператора является константным выражением, и это выражение имеет значение `true`.

Конечная точка условного оператора считается доступной, если доступна конечная точка хотя бы одного из ее доступных вложенных операторов. Кроме того, конечная точка условного оператора, не содержащего ключевого слова `else`, является доступной, если аргумент этого оператора не является константным выражением или это выражение имеет значение `false`.

Оператор-переключатель

Оператор-переключатель используется для выбора одного из путей в зависимости от значения своего аргумента. Оператор-переключатель состоит из ключевого слова `switch`, за которым следуют круглые скобки, в которых размещается аргумент оператора, и блока, в котором могут располагаться секции переключателя. Каждая секция переключателя содержит одну или несколько меток переключателя и список операторов. Метка переключателя состоит из ключевого слова `case`, за которым следует константное выражение и двоеточие, или ключевого слова `default`, за которым следует двоеточие.

Аргумент оператора-переключателя может иметь тип `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string` или перечислимый тип. Если аргумент имеет один из пользовательских типов, то для него должно быть определено единственное возможное неявное преобразование к одному из перечисленных выше типов, кроме перечислимого. Если такое преобразование отсутствует или возможно использование более одного неявного преобразования, то компилятор выдает сообщение об ошибке.

Константное выражение, стоящее в метке переключателя после ключевого слова `case`, должно иметь тип, имеющий неявное преобразование к типу аргумента оператора-переключателя. Если выражения двух или более меток одного оператора переключателя имеют одинаковые значения, компилятор выдает сообщение об ошибке.

Оператор-переключатель может иметь только одну метку `default`, управление по которой передается в том случае, если оно не может быть передано ни по одной другой метке.

Оператор-переключатель выполняется следующим образом:

- вычисляется значение выражения аргумента и преобразуется к одному из перечисленных выше типов;
- если полученное значение совпадает со значением выражения в одной из меток переключателя, управление передается первому оператору в последовательности операторов, следующих за данной меткой;
- если полученное значение не совпадает со значением выражения ни в одной из меток переключателя, и данный оператор-переключатель имеет метку `default`, то управление передается первому оператору в последовательности операторов, следующих за данной меткой;
- если полученное значение не совпадает со значением выражения ни в одной из меток переключателя, и данный оператор-переключатель не имеет метки `default`, то управление передается конечной точке оператора-переключателя.

В отличие от C и C++ в C# конечная точка последовательности операторов секции не может быть доступной. Обычно для обеспечения этого требова-

ния используется оператор `break`. В C и C++ это было настоятельной рекомендацией, но прямого запрета не было, и следующий фрагмент программы не содержал ошибок:

```
// Текст содержит ошибки
int    i = 0;
switch( n )
{
       case 7:i += a[7];
       case 6:i += a[6];
       case 5:i += a[5];
       case 4:i += a[4];
       case 3:i += a[3];
       case 2:i += a[2];
       case 1:i += a[1];
       case 0:i += a[0];
}
```

Подобные фрагменты программ позволяли повысить быстродействие ее критических фрагментов. Для реализации подобных конструкций в C# необходимо использовать операторы перехода, как это показано в следующем примере:

```
// Исправленный текст
int    i = 0;
switch( n )
{
       case 7:i += a[7];
              goto case 6;
       case 6:i += a[6];
              goto case 5;
       case 5:i += a[5];
              goto case 4;
       case 4:i += a[4];
              goto case 3;
       case 3:i += a[3];
              goto case 2;
       case 2:i += a[2];
              goto case 1;
       case 1:i += a[1];
              goto case 0;
       case 0:i += a[0];
}
```

Остается только надеяться, что в компиляторе C# используется хороший оптимизатор кода, и введение лишних операторов не скажется на быстродействии критического фрагмента программы.

Как уже говорилось выше, секция переключателя может иметь несколько меток. Поскольку все они относятся к одной последовательности операторов, на них не распространяется правило недостижимости конечной точки последовательности:

```
switch( n )
{
        case 7:i = a[7];
               break;
        case 6:
        case 5:i = a[6] + a[5];
               break;
        case 4:
        case 3:
        case 2:i = a[4] + a[3] + a[2];
               goto case 1;
        case 1:;
        default:      i = 0;
}
```

В этом примере нет никакой необходимости устанавливать метку `case 1`, но ее установка не приводит к ошибке.

Разработчики C# считают, что они облагодетельствовали человечество, оградив его от ошибок, возникавших при работе с оператором-переключателем в C и C++. Однако если уж они такие заботливые, не проще было бы вместо установки запретов автоматически устанавливать в конце каждого сегмента переключателя неявный оператор `break`. Кстати, это требование не распространяется на последнюю секцию блока.

Для того чтобы сделать конечную точку последовательности операторов недоступной, могут использоваться любые операторы с недоступной конечной точкой. Например, специалисты Microsoft советуют использовать для этого оператор `while`, аргументом которого является значение `true`. При этом во вложенном операторе должно вызываться исключение или использоваться другой оператор передачи управления, исключающий зависание программы в бесконечном цикле. Менее экзотическим способом сделать конечную точку последовательности операторов недоступной, является использование операторов `throw` или `return`:

```
int Fact(uint i )
{
```

```
        switch( i )
        {
                case 0:throw new ArgumentException();
                case 1:return 1;
                default: return i*Fact( i- 1);
        }
}
```

Конечно, факториал нуля равен единице, но в данном случае нужно было продемонстрировать возможность вызова исключения.

В операторе-переключателе используются те же правила сравнения, что и в других операторах и выражениях C#. Поэтому сравнение строк в этом операторе зависит от регистра используемых символов (это вполне естественно, поскольку для кодирования заглавных и строчных букв используются разные значения). Для устранения этого недостатка в аргументе оператора-переключателя вся строка может быть переведена в один регистр (верхний или нижний), а все выражения в метках должны быть записаны в этом регистре. Если аргументом оператора-переключателя является строка, то одно из выражений в метках может иметь значение `null`.

В блоке секций переключателя могут объявляться локальные переменные и константы. Область действия этих переменных и констант распространяется от места их объявления до конца блока. То есть эти переменные и константы объявляются не только для своей последовательности операторов, но и для всех следующих за ней последовательностей операторов. Это довольно необычно, поэтому заслуживает особого упоминания. Таким образом, все переменные в блоке имеют одну область определения независимо от того, в какой секции они определены или используются.

Секция оператора-переключателя является доступной, если доступен сам оператор-переключатель и выполняется, по крайней мере, одно из приведенных ниже условий:

- аргумент оператора-переключателя не является константным выражением;
- аргумент оператора-переключателя является константным выражением, но его значение совпадает со значением выражения в любой из меток данной секции;
- аргумент оператора-переключателя является константным выражением, значение которого не совпадает со значением выражения в любой из меток других секций, и данная секция имеет метку `default`;
- метка данной секции содержится в операторе `goto`, расположенном в любой другой доступной секции.

Конечная точка оператора-переключателя является доступной, если он доступен и выполняется хотя бы одно из перечисленных ниже условий:

- в любой из доступных секций данного оператора расположен любой оператор, завершающий выполнение этой секции и оператора-переключателя;
- аргумент оператора-переключателя не является константным выражением, и он не содержит секции с меткой `default`;
- аргумент оператора-переключателя является константным выражением, значение которого не совпадает со значением выражения ни в одной из меток секций данного оператора, и он не содержит секции с меткой `default`;
- секция с меткой `default` является последней секцией блока.

Одной из приятных особенностей C# является то, что в нем в качестве меток оператора-переключателя могут использоваться строковые переменные, что позволяет существенно повысить читабельность программы. Как, например, в приведенной ниже функции распределения задач:

```
public void Dispatch(string Task)
{
        switch(Task)
        {
                case "Add": Add();
                        break;
                case "Remove": Remove();
                        break;
                case "Process": Process ();
                        break;
                default: Idle()
                        break;
        }
}
```

Использование оператора-переключателя не только удобнее, но и эффективнее использования вложенных условных операторов, поскольку в нем применяется более эффективный алгоритм сравнения. Если число блоков в операторе-переключателе невелико, при его работе используется внутренняя таблица строковых констант программы, обеспечивающая соответствие всех экземпляров строки в программе одному и тому же объекту. Если заданная строка отсутствует в таблице, выполняется блок `default`, в противном случае выполняется блок `case` с соответствующей меткой. Если число блоков в операторе-переключателе велико, он создает собственную хэш-таблицу и производит в ней поиск.

Операторы цикла

Операторы цикла позволяют многократно выполнять свой вложенный оператор, проверяя при этом некоторые условия. Различают следующие операторы цикла:

- оператор `while`;
- оператор `do`;
- оператор `for`;
- оператор `foreach`.

В операторах цикла могут использоваться операторы `break` и `continue`. Оператор `break` передает управление конечной точке внешнего оператора, во вложенном операторе которого он вызван. Если синтаксис внешнего оператора не предусматривает совместной работы с оператором `break` (это относится, например, к условному оператору), считается, что оператор `break` вызван на уровне внешнего оператора, то есть он передает управление конечной точке внешнего оператора следующего уровня.

Оператор `continue` действует аналогично оператору `break`, однако, он передает управление не конечной точке внешнего оператора, а конечной точке вложенного оператора. В случае операторов цикла это различие весьма существенно.

Оператор while

Оператор `while` состоит из ключевого слова `while`, логического выражения, помещенного в круглые скобки, и следующего за ним вложенного оператора.

Оператор `while` выполняется следующим образом:

- вычисляется значение логического выражения;
- если логическое выражение имеет значение `false`, управление передается конечной точке оператора `while`;
- если логическое выражение имеет значение `true`, управление передается вложенному оператору;
- после достижения конечной точки вложенного оператора управление передается внешнему оператору `while`, выполнение которого не зависит от того, получил ли он управление от предыдущего оператора или от своего вложенного оператора.

Вложенный оператор оператора `while` доступен в том случае, если доступен сам оператор `while`, и в качестве его аргумента не использовано константное выражение, имеющее значение `false`.

Конечная точка оператора `while` доступна в том случае, если доступен сам оператор `while` и выполняется хотя бы одно из следующих условий:

- в качестве его аргумента не использовано константное выражение, имеющее значение `true`;
- вложенный блок оператора `while` имеет доступный оператор `break`, прерывающий его выполнение.

Ниже приведен пример использования оператора `while`:

```
int[]  a = { 1, 3, 0, 2, 4};
int           i = 0;

while( i < a.Length )
{
       if( a[i++] == 0)
       {
              break;
       }
}
```

Оператор do

Оператор `do` представляет собой, по сути, оператор `while`, в котором проверка логического условия осуществляется не до, а после выполнения вложенного блока. Оператор `do` состоит из ключевого слова `do`, вложенного оператора, ключевого слова `while` и логического выражения, помещенного в круглые скобки.

Оператор `do` выполняется следующим образом:

- управление передается вложенному оператору;
- вычисляется значение логического выражения;
- если логическое выражение имеет значение `false`, управление передается конечной точке оператора `do`;
- если логическое выражение имеет значение `true`, управление передается вложенному оператору.

Вложенный оператор оператора `do` доступен в том случае, если доступен сам оператор `do`.

Конечная точка оператора `do` доступна в том случае, если доступен сам оператор `do` и выполняется хотя бы одно из следующих условий:

- доступна конечная точка вложенного оператора и в качестве аргумента оператора `do` не использовано константное выражение, имеющее значение `true`;
- вложенный блок оператора `do` имеет доступный оператор `break`, прерывающий его выполнение.

Ниже приведен пример использования оператора `do`:

```
int[]  a = { 1, 3, 0, 2, 4};
int           i = 0;
```

```
do
{
        if( a[i++] == 0)
        {
                break;
        }
} while( i < a.Length );
```

Оператор for

Оператор `for` используется для инициализации, проверки и модификации условий цикла. Оператор `for` состоит из ключевого слова `for`, круглых скобок, в которых могут размещаться три выражения, отделяемые друг от друга точкой с запятой, и вложенного оператора.

Все три выражения оператора `for` являются необязательными, но при опускании любого из первых двух выражений необходимо оставлять стоящую за ним точку с запятой.

Первое выражение оператора `for` вызывается при передаче ему управления. Это выражение может содержать объявление локальных переменных или список операторов-выражений, разделяемых запятой. Как правило, это выражение используется для инициализации используемых в цикле переменных. В отличие от C++, где область действия переменной, определенной в первом выражении оператора `for`, распространяется от места ее объявления до конца блока, в котором помещен данный оператор, в C# область действия этой переменной распространяется от места ее объявления до конца вложенного блока оператора `for`, включая заголовок данного оператора.

Второе выражение является логическим выражением, и его значение оценивается перед каждой передачей управления вложенному оператору. Управление вложенному блоку передается только в том случае, если это выражение имеет значение `true`. В противном случае управление передается конечной точке оператора `for`.

Третье выражение вызывается после достижения вложенным оператором своей конечной точки. Как правило, в этом выражении производится модификация значений счетчика цикла и связанных с ним переменных.

Оператор `for` выполняется следующим образом:

- при передаче управления оператору `for` производится объявление и инициализация переменных, содержащихся в его первом выражении;
- производится вычисление значения второго выражения. Если это выражение отсутствует, считается, что оно имеет значение `true`;
- если логическое выражение оператора `for` имеет значение `false`, управление передается конечной точке данного оператора;

- если логическое выражение оператора `for` имеет значение `true`, управление передается вложенному оператору;
- при достижении конечной точки вложенного оператора вычисляется третье выражение оператора `for`. После этого снова вычисляется и проверяется значение второго выражения. Действия, предпринимаемые по результатам этой проверки, описаны выше.

Вложенный оператор `for` доступен в том случае, если доступен сам оператор `for` и выполняется одно из следующих условий:

- в операторе `for` отсутствует логическое выражение (второе выражение);
- логическое выражение оператора `for` не является константным логическим выражением, имеющим значение `false`.

Конечная точка оператора `for` доступна в том случае, если доступен сам оператор `for` и выполняется хотя бы одно из следующих условий:

- вложенный блок оператора `for` имеет доступный оператор `break`, прерывающий его выполнение;
- логическое выражение оператора `for` не является константным логическим выражением, имеющим значение `true`.

Ниже приведен пример использования оператора `for`:

```
int[]  a = { 1, 3, 0, 2, 4};
int            i;

for(i=0; i < a.Length; i++)
{
        if( a[i] == 0)
        {
                break;
        }
}
```

Оператор foreach

Оператор `foreach` используется для последовательного доступа к элементам коллекции, для каждого из которых выполняется вложенный оператор. Оператор `foreach` состоит из ключевого слова `foreach` и круглых скобок, содержащих объявление переменной цикла, ключевое слово `in` и выражение. За круглыми скобками располагается вложенный оператор.

Переменная цикла обязательно должна использоваться только для чтения, и ее область действия ограничена вложенным оператором, в нем она представляет собой элемент коллекции, над которым производится операция.

Переменной цикла нельзя присваивать значение или передавать ее в аргументах, имеющих модификаторы `ref` или `out`. Если в программе будет произведено одно из перечисленных действий, компилятор выдаст сообщение об ошибке.

Тип выражения, стоящего за ключевым словом `in`, должен иметь тип коллекции и явное преобразование к типу переменной цикла. Для того чтобы тип являлся типом коллекции, он должен удовлетворять следующим требованиям:

- содержать метод `GetEnumerator()`, имеющий модификатор права доступа `public` и возвращающий значение типа структуры, класса или интерфейса;
- тип структуры, класса или интерфейса, возвращаемый методом `GetEnumerator()`, должен содержать метод `MoveNext()`, имеющий модификатор права доступа `public` и возвращающий значение типа `bool`;
- кроме того, этот тип структуры, класса или интерфейса должен содержать доступное для чтения свойство с именем `Current`. Тип данного свойства является типом элемента для типа коллекции.

Тип `System.Array` является типом коллекции. Поскольку все массивы являются потомками класса `System.Array`, для них может использоваться оператор `foreach`. Просмотр массивов производится в порядке возрастания индекса. Индексы многомерного массива возрастают справа налево.

Оператор `foreach` выполняется следующим образом:

- Вычисляется значение выражения для получения объекта, имеющего тип коллекции `Collection_Type`. Если коллекция имеет тип ссылки, а полученный объект имеет значение `null`, вызывается исключение `NullReferenceException`.
- Для получения счетчика `Enum` вызывается метод `Collection_Type.GetEnumerator()`. Если счетчик имеет тип ссылки, а полученный объект имеет значение `null`, вызывается исключение `NullReferenceException`. Счетчик является внутренней переменной цикла, программный доступ к которой закрыт.
- Вызывается метод `Enum.MoveNext()`.
- Если метод возвращает значение `false`, управление передается конечной точке оператора `foreach`.
- Если метод возвращает значение `true`, значение свойства `Enum.Current` преобразуется к типу переменной цикла и сохраняется в ней.
- Управление передается вложенному оператору.
- При достижении конечной точки вложенного оператора снова вызывается метод `Enum.MoveNext()`, проверяется возвращаемое им значение, и производятся описанные выше действия.

Вложенный оператор `foreach` доступен в том случае, если доступен сам оператор `foreach`. Аналогично конечная точка вложенного оператора `foreach` доступна в том случае, если доступен сам оператор `foreach`.

Ниже приведен пример использования оператора `foreach`:

```
int[]  a = { 1, 3, 0, 2, 4};

foreach( int i in a )
{
        if( i == 0)
        {
                break;
        }
}
```

Операторы перехода

Операторы перехода используются для безусловной передачи управления в другую точку программы. Различают следующие операторы перехода:

- оператор `break`;
- оператор `continue`;
- оператор `goto`;
- оператор `return`;
- оператор `throw`.

Точка программы, в которую осуществляется переход, определяется назначением используемого оператора. Если оператор перехода находится в блоке, а точка программы, в которую осуществляется переход, расположена за пределами этого блока, то говорят, что оператор перехода используется для выхода из блока. Следует помнить, что в C# операторы перехода могут передавать управление только в пределах того блока, в котором он определен, и во внешний блок, но ни один из них не может передать управление во вложенный блок.

Конечная точка всех операторов перехода недостижима.

Выполнение операторов перехода существенно изменяется, если они расположены в блоках оператора `try`. Если при выполнении этих операторов не пересекаются границы блоков оператора `try`, операторы перехода просто осуществляют выход из блока или передают управление по указанной метке. Если же оператор перехода используется для выхода из одного или более блоков `try`, имеющих связанные с ними блоки `finally`, то сначала управ-

ление передается блоку `finally` первого блока `try`, из которого производится выход (внутреннего блока). При достижении конечной точки этого блока `finally` управление передается блоку `finally` следующего блока `try`, из которого производится выход (внешнего по отношению к предыдущему). Процесс будет продолжаться до тех пор, пока не будут выполнены блоки `finally` всех блоков `try`, из которых производится выход. И только после этого управление будет передано по назначению. Это продемонстрировано на следующем примере:

```
void Some_Func()
{
        while (true)
        {
                try
                {
                        try {
                                Console.WriteLine("Шаг цикла ");
                                break;
                        }
                        finally
                        {
                                Console.WriteLine("Внутренний блок try");
                        }
                }
                finally
                {
                        Console.WriteLine("Внешний блок try");
                }
        }
        Console.WriteLine("Выход из цикла");
}
```

В результате выполнения этого метода каждая из строк будет выведена на экран один раз в порядке их расположения в программе.

Оператор break

Оператор `break` используется для выхода из вложенных блоков операторов цикла и оператора-переключателя. Оператор `break` состоит из ключевого слова `break` и точки с запятой.

Необходимо помнить о том, что данный оператор используется не только в операторах цикла, но и в операторе-переключателе. Дел в том, что обычно

вызов оператора break в операторе цикла приводит к выходу из внутреннего цикла независимо от того, во вложенном блоке какого оператора он был вызван. То есть вызов этого оператора может привести к выходу из нескольких вложенных блоков, если все блоки, кроме последнего, не были блоками оператора цикла. Однако если один из этих блоков является блоком оператора-переключателя, то вызов оператора break приведет только к выходу из оператора-переключателя.

Обнаружив, что оператор break вызван в блоке, не вложенном ни в один оператор цикла или оператор-переключатель, транслятор выдает сообщение об ошибке. То же самое происходит и в том случае, когда с использованием оператора break пытаются выйти из блока finally, независимо от того, заключен он во вложенный блок оператора цикла или нет.

Оператор continue

Оператор continue используется для передачи управления конечной точке вложенного оператора в операторах цикла. Если этот оператор вызывается не во вложенном операторе оператора цикла, то компилятор выдает сообщение об ошибке. Оператор continue состоит из ключевого слова continue и точки с запятой.

Если вложенный оператор оператора цикла содержит блок finally, и оператор continue вызывается из этого блока, то компилятор выдает сообщение об ошибке.

Оператор goto

Оператор goto используется для передачи управления помеченному оператору. Оператор goto состоит из ключевого слова goto, метки и точки с запятой. В качестве метки данного оператора могут выступать как простые метки, так и метки оператора-переключателя.

Если метка, указанная в операторе goto, отсутствует или находится вне области действия данного оператора, компилятор выдает сообщение об ошибке. Область действия оператора goto распространяется на блок, в котором он вызван, и на внешние блоки. Этот оператор может использоваться для выхода из вложенных блоков оператора, причем уровень вложенности не имеет для него значения. Если в качестве метки оператора goto указана метка оператора-переключателя, то необходимо, чтобы одним из внешних блоков был бы блок оператора-переключателя. Кроме того, требуется, чтобы этот блок содержал указанную метку. Единственным исключением из этого правила является метка case, для которой требуется, чтобы использованное в ней выражение имело неявное преобразование к типу аргумента оператора-переключателя, а значение этого выражения совпадало со значением выражения, использованного в одной из меток case данного оператора.

Если оператор goto расположен в блоке finally, то и метка, по которой осуществляется переход, должна располагаться в том же блоке. В противном случае транслятор выдаст сообщение об ошибке.

Оператор return

Оператор return используется для выхода из функции. Оператор return состоит из ключевого слова return и точки с запятой. После ключевого слова return может располагаться выражение, тип которого имеет неявное преобразование к типу возвращаемого значения функции. Значение этого выражения представляет собой значение, возвращаемое функцией, завершающей работу по данному оператору return.

Выражение в операторе return отсутствует в методах, имеющих тип возвращаемого значения void, конструкторах, деструкторах, а также в процедурах доступа set свойств и индексаторов. Выражение в операторе return присутствует в методах, имеющих тип возвращаемого значения, отличный от void, перегруженных операциях и в процедурах доступа get свойств и индексаторов. Оператор return не может располагаться в блоках finally.

Оператор return выполняется следующим образом:

- если оператор return содержит выражение, то сначала вычисляется значение этого выражения и преобразуется к типу возвращаемого значения метода, используя для этого неявное преобразование типов;
- если оператор return вызывается внутри одного или более блоков try, имеющих связанные с ними блоки finally, сначала управление передается блоку finally первого блока try, из которого производится выход (внутреннего блока). При достижении конечной точки этого блока finally управление передается блоку finally следующего блока try, из которого производится выход (внешнего по отношению к предыдущему). Процесс будет продолжаться до тех пор, пока не будут выполнены блоки finally всех блоков try, из которых производится выход;
- управление передается функции, вызвавшей данный метод, и возвращает при этом значение выражения, вычисленное в операторе return.

Оператор throw

Оператор throw используется для вызова исключения. Оператор throw состоит из ключевого слова throw и точки с запятой. После ключевого слова throw может располагаться выражение. Это выражение должно иметь тип класса System.Exception или тип класса, производного от System.Exception. Если данное выражение имеет значение null, то вызывается исключение NullReferenceException.

Выражение в операторе `throw` может отсутствовать только в том случае, если этот оператор вызывается в обработчике `catch` для передачи обрабатываемого в нем исключения на последующую обработку.

При вызове исключения управление передается первому обработчику `catch` оператора `try`, который может обработать данный тип исключения. При поиске соответствующего обработчика исключения выполняются следующие действия:

- Если оператор `try`, во вложенном блоке которого вызвано исключение, имеет обработчики `catch`, производится просмотр типов исключений, обрабатываемых данными блоками. Если найден обработчик `catch`, обрабатывающий тип вызванного исключения или тип его базового класса, управление передается этому обработчику.
- Если оператор `try`, во вложенном блоке которого вызвано исключение, не содержит обработчиков `catch` или ни один из них не обрабатывает исключения данного типа, исключение передается внешнему оператору `try`. Для исключения появления необработанных исключений все приложение автоматически помещается в оператор `try`, обрабатывающий все типы исключений и выдающий сообщение пользователю о том, что в его приложении не обработано исключение.
- Если оператор `try`, передающий исключение внешнему оператору, имеет блок `finally`, то управление сначала передается этому блоку. Если достигается конечная точка этого блока, то внешнему оператору передается исходное исключение. Если же в блоке `finally` вызывается новое исключение, то старое исключение уничтожается и вместо него передается новое исключение.

При переходе к внешним операторам `try` последовательно осуществляется выход из всех вложенных блоков и вызванных функций данного внешнего оператора. Если в потоке или в процессе, в котором было вызвано исключение, отсутствуют его обработчики, то этот поток или процесс прекращает свою работу. Способ прекращения работы потока или процесса определяется их реализацией.

Оператор try

Оператор `try` позволяет создать структуры для обработки исключений, возникающих в процессе выполнения программы в их блоке. Он позволяет задать свой обработчик для каждого возможного типа исключений или пропустить определенные типы исключений для обработки на более высоком уровне.

Оператор `try` состоит из ключевого слова `try` и блока. За блоком обязательно должен следовать хотя бы один блок. Это может быть блок обработчика `catch` или блок `finally`. Обработчики `catch` и блок `finally` могут исполь-

зоваться как совместно, так и по отдельности, причем ни один обработчик catch не может располагаться за блоком finally.

Обработчик catch состоит из ключевого слова catch, круглых скобок, в которых указан тип обрабатываемого исключения, и блока, содержащего обработчик данного исключения. За типом обрабатываемого исключения в круглых скобках может быть указан идентификатор, используемый для доступа к объекту исключения в его обработчике.

Тип исключения, обрабатываемого в блоке обработчика catch, должен представлять собой класс System.Exception или производный от него класс. Указание в качестве типа обрабатываемого исключения класса System.Exception, приводит к тому, что этот обработчик будет использован для всех типов исключений. Поскольку поиск обработчика исключения ведется последовательно, подобный обработчик следует помещать последним в списке. Для простоты, в том случае, когда в данном обработчике не требуется доступ к объекту данного исключения, в нем разрешается опускать круглые скобки с типом обрабатываемого исключения, то есть заменять блок

```
catch (System.Exception)
{
        // Обработка исключения
}
```

блоком

```
catch
{
        // Обработка исключения
}
```

Поскольку эти блоки эквивалентны, то их нельзя использовать в одном операторе try.

Типы исключений могут располагаться только в порядке, обратном наследованию. То есть блок обработки любого типа исключения не может располагаться за блоком обработки базового типа данного исключения. Это требование логично, поскольку подобное расположение сделает недоступным обработчик производного типа. В принципе это тянет только на предупреждение, однако в данном случае это считается ошибкой, и компилятор соответствующее предупреждение выдает.

Идентификатор, определенный в заголовке обработчика catch, может использоваться только для чтения, и его область действия ограничена блоком соответствующего обработчика catch. При попытке присвоить данной переменной какое-нибудь значение или передать ее в аргументе с модификатором ref или out транслятор выдаст сообщение об ошибке. Использование

этого идентификатора является единственной возможностью получить доступ к обрабатываемому исключению в блоке обработчика `catch`.

Если обработка исключения не заканчивается в данном обработчике `catch`, он должен содержать оператор `throw`, не содержащий выражения. Этот оператор вызывает исключение для внешнего оператора `try`.

Для выхода из блока `finally` не могут использоваться операторы `break`, `continue` и `goto`. Если эти операторы вызываются в блоке `finally`, то они должны передавать управление только в пределах данного блока. В блоке `finally` не может вызываться оператор `return`.

Оператор `try` выполняется следующим образом:

- Управление передается блоку оператора `try`.
- Если управление передается конечной точке блока оператора `try` и этот оператор содержит блок `finally`, управление передается блоку `finally`. В противном случае управление передается конечной точке оператора `try`.
- Если при выполнении блока оператора `try` вызывается исключение, производится проверка его обработчиков `catch` на наличие среди них обработчика данного исключения. Исключение передается тому обработчику `catch`, в круглых скобках которого указан тип данного исключения или тип базового класса данного исключения.
- Если в операторе `try` отсутствуют обработчики `catch` или ни один из них не обрабатывает тип вызванного исключения, управление передается блоку `finally`, если таковой имеется, а затем — внешнему оператору `try`. Если внешний оператор `try` отсутствует, управление передается конечной точке данного оператора `try`.
- Если в заголовке обработчика `catch`, которому было передано управление, указан идентификатор, то в него помещается ссылка на обрабатываемый объект исключения.
- Управление передается блоку обработчика `catch`.
- Если управление передается конечной точке блока обработчика `catch` и соответствующий оператор `try` имеет блок `finally`, то управление передается блоку `finally`, в противном случае оно передается конечной точке оператора `try`.

Содержимое блока `finally` выполняется при любой передаче управления конечной точке оператора `try`. Это происходит при нормальном завершении работы последовательности операторов в блоке `try`, при выходе из него с использованием операторов перехода или при вызове исключения. Единственной возможностью обойти блок `finally` может быть вызов исключения в обработчике `catch`.

Единственным случаем, когда конечная точка блока `finally` становится недостижимой, является вызов исключения в этом блоке. Если блок `finally`

был вызван в процессе передачи другого исключения, то его заменяет исключение, вызванное в блоке `finally`.

Если доступен сам оператор `try`, то доступны его блок, обработчики `catch` и блок `finally`.

Конечная точка оператора `try` доступна, если одновременно выполняются два условия:

1. Доступна конечная точка блока оператора `try` или конечная точка по крайней мере одного из обработчиков `catch`.
2. Если присутствует блок `finally`, его конечная точка должна быть доступной.

Операторы checked и unchecked

Операторы `checked` и `unchecked` используются для включения или отключения проверки на переполнение при выполнении целочисленных операций и преобразований. Оператор `checked` состоит из ключевого слова `checked` и блока, на который распространяется его действие. Аналогично оператор `unchecked` состоит из ключевого слова `unchecked` и блока, на который распространяется его действие.

Оператор `checked` устанавливает режим проверки на переполнение. Если переполнение возникает в процессе вычисления константного выражения, расположенного в его блоке, то компилятор выдает сообщение об ошибке. Если вызвавшая переполнение операция была исполнена в процессе выполнения программы, вызывается исключение `OverflowException`.

Оператор `unchecked` отключает режим проверки на переполнение. В его блоке возникает переполнение, в полученном результате отбрасываются все старшие биты, которые не могут быть сохранены в формате результата.

Оба этих оператора оказывают влияние на результаты следующих операций:

- предопределенных унарных операций `++` и `--`, производимых над целочисленными операндами;
- предопределенной унарной операции `–`, производимой над целочисленными операндами;
- предопределенных бинарных операций `+`, `–`, `*` и `/`, если оба их операнда являются целочисленными величинами;
- при явном преобразовании значения из одного целочисленного типа в другой.

Операторы `checked` и `unchecked` аналогичны операциям `checked` и `unchecked`. Единственное различие между ними состоит в том, что операторы оказывают воздействие на блоки, а операции — на выражения.

Оператор lock

Оператор `lock` осуществляет взаимно исключающую блокировку указанного объекта, выполняет вложенный оператор и освобождает объект. Оператор `lock` состоит из ключевого слова `lock`, выражения, помещенного в круглые скобки, и вложенного оператора.

Выражение, используемое в качестве аргумента оператора `lock`, имеет тип ссылки. При указании в аргументе оператора `lock` выражения, имеющего тип значения, для него не производится неявное преобразование к типу ссылки. Поэтому использование такого выражения является ошибкой.

Оператор `lock` вида

```
lock ( Argument )
{
        // Обработка
}
```

эквивалентен следующей последовательности операторов

```
System.CriticalSection.Enter( Argument );
try {
        // Обработка
}
finally {
        System.CriticalSection.Exit( Argument );
}
```

за исключением того, что выражение `Argument` в нем вычисляется однократно. Операции, выполняемые методами `Enter` и `Exit` класса `System.CriticalSection`, зависят от его реализации.

Обычно в качестве объекта, взаимно исключающего доступ статических методов класса, используется объект `System.Type`, как это показано в приведенном ниже примере:

```
class Demo
{
        object obj;
        public static void First()
        {
                lock (typeof(Demo))
                {
                        // Обработка
```

```
            }
        }
        public static void Second()
        {
            lock (typeof(Demo))
            {
                // Обработка
            }
        }
}
```

В этом примере функции `First` и `Second` не могут одновременно получить доступ к внутреннему объекту данного класса.

Глава 10

Классы

Класс представляет собой объект, содержащий данные (константы, поля и события), функции (методы, свойства, индексаторы, операции, конструкторы и деструкторы) и вложенные типы. Классы поддерживают наследование: механизм, позволяющий строить иерархии классов, в которых производный класс полностью реализует функциональные возможности базового класса, добавляя к нему новые функциональные возможности.

Реализация класса

Реализация класса состоит из ключевого слова `class`, идентификатора класса и тела класса. Перед ключевым словом `class` могут располагаться набор атрибутов и набор модификаторов класса. После идентификатора класса может располагаться спецификатор базового класса. За телом класса может стоять точка с запятой.

Модификаторы класса

В качестве модификаторов класса могут выступать ключевые слова:

`new`	`public`
`protected`	`internal`
`private`	`abstract`
`sealed`	

Каждый из этих модификаторов может появляться в заголовке класса только один раз. Модификатор `new` может использоваться только для вложенного класса. Он подавляет предупреждение, выдаваемое при скрытии члена базового класса (скрытие членов класса будет рассмотрено позднее). Если в классе, помеченном модификатором `new`, члены базового класса не скрываются, транслятор выдает предупреждение.

Модификаторы `public`, `protected`, `private` и `internal` определяют режим доступа к классу. Первые три из этих модификаторов используются и в языке программирования C++ и имеют то же самое назначение. Модификатор `internal` будет рассмотрен позднее.

Модификатор `abstract` указывает на то, что данный класс не может использоваться самостоятельно, а может только служить базовым для других классов. Основные особенности абстрактных классов перечислены ниже:

- нельзя создать статический или динамический объект абстрактного класса;
- в процессе работы приложения можно создать переменные и значения, имеющие тип абстрактного класса, однако, они содержат или значение `null`, или ссылку на объект обычного класса, являющегося потомком данного абстрактного класса;
- абстрактный класс может (но не должен) содержать абстрактные методы и процедуры доступа (accessors);
- абстрактный класс не может быть терминальным.

Потомок абстрактного класса должен перегружать все абстрактные методы и процедуры доступа базового абстрактного класса, как это показано в приведенном ниже примере:

```
abstract class Abstract_Class
{
        public abstract void Some_Func();
        public void Other_Func() {}
}
class Some_Class: Abstract_Class
{
        public override void F()
        {
                // Реализация функции
        }
}
```

Как видно из приведенного примера, перегружать требуется только абстрактные методы, хотя это не означает, что другие методы абстрактного класса не могут быть перегружены.

Модификатор `sealed` указывает на то, что данный класс является терминальным, то есть, он не может служить базовым для других классов. Как уже отмечалось выше, терминальный класс не может быть абстрактным. Этого следовало ожидать, поскольку абстрактный класс может использоваться только в качестве базового, а терминальный класс не может использоваться в этом качестве.

Использование терминальных классов позволяет несколько повысить быстродействие программы, поскольку в таких классах виртуальные функции могут быть преобразованы в обычные функции.

Спецификатор базового класса

Язык программирования C# не поддерживает множественного наследования классов, то есть каждый класс может иметь только один базовый класс. За именем базового класса, если оно указано, следует перечень реализуемых им интерфейсов. Интерфейсы будут рассмотрены в главе 11.

См. гл. 11.

Создаваемый класс не может расширять право доступа к базовому классу. Например, следующий текст программы содержит ошибку:

```
private class Private_Class
{
        // Реализация базового класса
}
protected class Protected_Class: Private_Class    // Ошибка
{
        // Реализация производного класса
}
```

В спецификаторе указывается класс, непосредственным потомком которого является описываемый класс. По умолчанию (то есть при отсутствии в спецификаторе имени базового класса) в качестве базового класса выбирается `object`. Поэтому все классы, кроме класса `object`, имеют свой базовый класс. Таким образом, все классы принадлежат одной иерархической структуре наследования, в корне которой расположен класс `object`.

Иерархическая структура не может содержать циклов. Это относится и к вложенным классам. Ниже приведены основные ошибки, связанные с наследованием классов:

```
class First: Third   // Ошибка!
{
        ...
}
class Second: First
{
        ...
}
class Third: Second
{
        ...
}
```

```
class First: Second.Third    // Ошибка!
{
        ...
}
class Second: First
{
        public class Third {}
}
```

Отыскание подобных ошибок затрудняется тем, что в C# в отличие от C++ порядок объявления классов жестко не задан, и базовые классы могут объявляться после производных от них классов. Поэтому в приведенном выше примере строка с ошибкой помечена условно, исходя из синтаксиса C++.

При объявлении классов допустима рекурсия, например, приведенный ниже пример полностью корректен:

```
class First
{
        class Second: First
        {
                ...
        }
}
```

Таким образом, класс `Second` содержит бесконечное число своих объектов.

Члены класса

В теле класса содержится описание членов класса, разделенных точкой с запятой. Членами класса являются:

- константы, содержащие постоянные значения, связанные с классом;
- поля, представляющие собой внутренние переменные класса;
- методы, реализующие функциональные возможности класса;
- свойства, определяющие именованные атрибуты и действия, связанные с чтением и записью данных атрибутов;
- события, определяющие извещения, посылаемые классом;
- индексаторы, позволяющие образовывать массивы из объектов класса;
- операции, определяющие операции, которые могут быть применены к объектам данного класса;

- конструкторы объектов, выполняющие операции, необходимые для создания объекта данного класса;
- деструкторы, выполняющие операции, которые необходимо произвести перед уничтожением объекта данного класса;
- статические конструкторы, выполняющие операции, необходимые для создания самого класса;
- определения типов, свойственных только данному классу.

Члены класса, выполняющие некоторую обработку, называются функциональными членами класса. К функциям класса относятся методы, свойства, индексаторы, операции, конструкторы и деструкторы класса.

Заголовок класса создает для тела класса новую область видимости имен. В этой области видимости имен действуют следующие соглашения:

- Конструкторы и деструкторы должны иметь то же имя, что и включающий их класс. Имена всех остальных членов класса должны отличаться от имени включающего их класса.
- Имя любой константы, поля, свойства, события или типа должно отличаться от имен всех других членов данного класса.
- Имена методов класса должны отличаться от имен других членов класса. Допустимо иметь несколько методов, имеющих одинаковое имя, но различный набор аргументов.
- Нельзя определить в классе два индексатора с одним и тем же именем и набором аргументов.
- Нельзя определить в классе две операции с одним и тем же именем и набором аргументов

Если в классе объявляется член, имеющий то же имя и набор аргументов, что и член базового класса, то соответствующий член базового класса скрывается.

Наследование

Класс наследует все члены базового класса. Это означает, что в него автоматически включаются все члены базового класса, кроме его конструктора и деструктора. Основные правила наследования приведены ниже:

- наследование транзитивно, то есть в новый класс включаются не только все члены, явным образом объявленные в базовом классе, но и унаследованные им члены своего базового класса;
- наследование расширяет функциональные возможности базового класса. В производном классе могут быть объявлены новые члены, но не могут быть удалены унаследованные члены;

- наследуются все члены базового класса, кроме конструкторов и деструкторов. Однако некоторые из унаследованных членов могут оказаться недоступными в производном классе вследствие установленного для них режима доступа;
- при объявлении в производном классе члена с тем же именем и набором переменных, что и в базовом классе, член базового класса скрывается . Скрытие не означает удаления члена базового класса — он просто становится недоступным в производном классе. При скрытии члена базового класса транслятор выдает сообщение об ошибке. Чтобы подавить это сообщение, необходимо использовать модификатор `new`;
- поскольку объект производного класса содержит все члены базового класса, поскольку тип класса относится к типам ссылок, возможно преобразование типа производного класса к типу базового класса;
- для реализации полиморфизма в классе могут быть объявлены виртуальные методы, свойства и индексаторы. Полиморфизм позволяет в рамках иерархии классов иметь несколько версий одной и той же функции. Решение о том, какая именно версия данной функции будет использована в данном конкретном случае, определяется в процессе выполнения приложения.

Модификаторы режима доступа

Кроме используемых в C++ модификаторов доступа `public`, `protected` и `private`, в C# определен новый модификатор прав доступа `internal`, используемый для типов. Этот модификатор ограничивает область действия типа файлами одной сборки (assembly). Все модификаторы доступа должны использоваться раздельно. Единственным исключением является сочетание прав доступа `protected internal`.

При отсутствии модификатора прав доступа соответствующий член класса по умолчанию имеет право доступа `private`.

Поскольку все типы в C# являются объектами, все они имеют связанные с ними права доступа. При этом действует правило, что типы члена класса (к ним относится тип самого члена класса, аргументы и возвращаемые значения функций) должны иметь право доступа не строже, чем право доступа самого члена класса. То есть, функция, имеющая модификатор права доступа `public`, не может возвращать значение типа, имеющего модификатор права доступа `protected`.

Статические и нестатические члены класса

Член класса может быть *статическим* и *нестатическим*. Основное отличие между ними заключается в том, что в каждом объекте класса существует своя копия нестатического члена, но все объекты класса разделяют одну и туже копию статического члена класса. Для удобства можно считать, что статические члены принадлежат самому классу, а нестатические — его объектам.

Для объявления статического члена класса используется модификатор `static`. Статическими могут быть поля, методы, свойства, операции или конструкторы класса. Поскольку константы и объявления типов могут быть только статическими членами класса, при их объявлении модификатор `static` не указывается.

Ниже приведены основные особенности работы со статическими членами класса:

- для обеспечения доступа к статическому члену класса с использованием конструкции `Some_Type.Member` элемент `Some_Type` должен представлять собой имя класса. Использование в данном контексте имени объекта класса является ошибкой;
- независимо от того, сколько было создано объектов данного класса, создается только один экземпляр статического элемента;
- поскольку статическая функция (метод, свойство, индексатор, конструктор или деструктор) не исполняется в конкретном объекте данного класса, при их вызове не может быть использовано ключевое слово `this`.

Если при объявлении члена класса опущен модификатор `static`, то этот член класса является нестатическим. Исключением из этого правила являются упомянутые выше объявления констант и типов.

Ниже приведены основные особенности работы со статическими членами класса:

- для обеспечения доступа к нестатическому члену класса с использованием конструкции `Some_Obj.Member` элемент `Some_Obj` должен представлять собой имя объекта класса. Использование в данном контексте имени класса является ошибкой;
- каждый объект класса использует свои независимые копии всех нестатических полей класса.

Ниже приведен пример использования статических и нестатических членов класса:

```
class Demo
{
              byte   i;
       static byte   j;

       void Some_Func()
       {
              i = 0;         // Все в порядке
              j = 0;         // Все в порядке
       }
```

```
        static void Static_Func()
        {
                i = 0;      // Ошибка! Обращение к нестатическому элементу
                j = 0;      // Все в порядке
        }

        static void Main()
        {
                Demo   d      = new Demo();

                d.i    = 0;   // Все в порядке
                d.j    = 0;   // Некорректный вызов
                Demo.i = 0;   // Некорректный вызов
                Demo.j = 0;   // Все в порядке
        }
}
```

Константы

Константы представляют собой статические члены класса, содержащие постоянные значения, однако, они не требуют и не допускают использования модификатора `static`. Объявление констант состоит из ключевого слова `const`, за которым следует тип константы и список констант, разделяемых запятой. В конце объявления констант ставится точка с запятой. Перед ключевым словом `const` может размещаться набор атрибутов и набор модификаторов констант. Элемент списка констант представляет собой имя константы, за которым следует знак присваивания и значение константы.

В качестве модификаторов констант могут использоваться следующие ключевые слова:

`new`	`public`
`protected`	`internal`
`private`	

Константы могут иметь тип: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, а также перечисляемый тип и тип ссылки. При инициализации константы должно использоваться значение того же типа, что и инициализируемая константа, или другого типа, для которого существует явное преобразование к типу константы. Право доступа типа константы не должно быть жестче, чем право доступа к самой константе.

Константы представляют собой и могут являться составной частью константного выражения. Это позволяет использовать их в качестве меток при

объявлении переменной перечисляемого типа в качестве элемента этой переменной и в других случаях использования константных выражений.

Константное выражение представляет собой любое выражение, значение которого может быть определено в процессе трансляции. Поскольку единственным типом ссылки, переменная которого может иметь ненулевое значение в процессе трансляции, является `string`, то все остальные переменные, имеющие тип ссылки и входящие в константное выражение, должны иметь значение `null`.

При инициализации констант допускается использование константных выражений, содержащих уже определенные до этого константы.

При необходимости создать константу, тип которой не может быть использован в константном выражении, или значение которой не может быть вычислено в процессе трансляции, вместо нее может быть создано поле, используемое только для чтения. Описание подобных полей будет приведено ниже.

В одной строке может быть объявлено любое число констант одного типа с одинаковым набором атрибутов и модификаторов. Число строк объявления констант одного типа с одинаковым набором атрибутов и модификаторов не ограничено. Ниже приведен пример объявления констант:

```
class Some_Class
{
        public const  int    i = 1, j = 2, k = 3;
        public const  bool   t = true;
        public const  bool   f = false;
}
```

В C# допускаются очень сложные зависимости элементов константных выражений в пределах одного проекта. Единственным ограничением является отсутствие в таких зависимостях циклических конструкций. Например, допустима следующая конструкция:

```
class Some_Class
{
        public const  int    i = 1, j = Other_Class.i + 1, k = 3;
        public const  bool   t = true;
        public const  bool   f = false;
}

class Other_Class
{
        public const  int    i = Some_Class.i + 1;
}
```

При этом в классе `Some_Class` ничего не изменится по сравнению с предыдущим примером.

Поля

Поля класса содержат его переменные. Объявление полей состоит из типа поля и списка идентификаторов полей, разделяемых запятой. В конце объявления полей ставится точка с запятой. В объявлении поля допускается его инициализация. Синтаксис инициализации поля аналогичен синтаксису инициализации константы. Перед типом поля может размещаться набор атрибутов и набор модификаторов полей.

В качестве модификаторов полей могут использоваться следующие ключевые слова:

```
new                                    public
protected                              internal
private                                static
readonly
```

Право доступа к типу поля не должно быть жестче, чем право доступа к самому полю.

В одной строке может быть объявлено любое число полей одного типа с одинаковым набором атрибутов и модификаторов. Число строк объявления полей одного типа с одинаковым набором атрибутов и модификаторов не ограничено. Ниже приведен пример объявления полей:

```
class Some_Class
{
        public         int    i, j = 2, k = 3;
        public         int    m, n, l;
        public static bool    t = true;
        public         bool   f;
}
```

Поля, используемые только для чтения

Модификатор `readonly` используется для создания внутренних констант класса. В отличие от обычных констант, рассмотренных выше, значение внутренней константы не может быть задано константным выражением, поэтому это значение определяется в заголовке класса или в его конструкторе. Во всех остальных функциях класса изменение значения этого поля недопустимо. Присваивание значения полю, имеющему модификатор `readonly`, разрешено только в следующих случаях:

- При объявлении поля (инициализация является составной частью объявления поля).
- Значение нестатическому полю может быть присвоено в конструкторе объекта данного класса. Значение статическому полю может быть при-

своено в статическом конструкторе данного класса. Поля, имеющие модификатор `readonly`, могут передаваться этим конструкторам в качестве аргумента, сопровождаемого модификатором `out` или `ref`. Передача этого поля в качестве аргумента, сопровождаемого модификатором `out` или `ref`, любой другой функции является ошибкой.

Если одновременно с модификатором `readonly` не указан модификатор `static`, в каждом объекте класса может использоваться собственное значение константы. В этом случае поле, строго говоря, не содержит константу, поскольку не является статическим.

Поля, имеющие модификатор `static readonly`, используются для создания констант, значения которых могут быть определены только в процессе выполнения приложения. Обычно это связано с тем, что при формировании значений констант вызываются некоторые функции. Ниже приведен пример определения подобных констант:

```
public class RGB
{
       public static readonly      RGB    Black         = new RGB(0, 0,
0);
       public static readonly      RGB    Dark_Gray     = new RGB(96,
96, 96);
       public static readonly      RGB    Light_Gray    = new RGB(192,
192, 192);
       public static readonly      RGB    White         = new RGB(255,
255, 255);

       privatebyte   Red;
       privatebyte   Green;
       privatebyte   Blue;

       public RGB(byte R, byte G, byte B)
       {
              Red = R;
              Green = G;
              Blue = B;
       }
}
```

Версии констант и статических полей, используемых только для чтения

Двоичные версии полей, используемых только для чтения, и констант имеют различную семантику. Это связано с тем, что значения констант вычисляются в процессе трансляции приложения, а значения любых полей — только в процессе работы приложения. Рассмотрим простейший пример,

содержащий два связанных между собой проекта, каждый из которых выполняется в своем пространстве имен.

```
namespace First_Project
{
        public class Demo
        {
                public static readonly      short  F      = 0;
                public        const         short  C      = 0
        }
}
namespace Second_Project
{
        class Demo
        {
                static void Main()
                {
                        Console.WriteLine( First_Project.Demo.F );
                        Console.WriteLine( First_Project.Demo.C );
                }
        }
}
```

Различие пространств имен проектов означает их раздельную компиляцию. После компоновки этих двух проектов на экран будут выводиться два нулевых значения. Изменим значения константы и поля в первом проекте на единичные и перекомпилируем только первый проект. В результате этого на экран будет выводиться единичное и нулевое значения. Это связано с тем, что, хотя поле и используется для хранения константы, ее значение определяется только в процессе выполнения приложения и не может быть определено в процессе компиляции другого проекта.

Инициализация полей

Если при объявлении поля не производится его явная инициализация, то его начальное значение определяется типом этого поля, исходя из нулевого значения всех его битов. При загрузке класса во все его статические поля записываются значения, определяемые статическим конструктором данного класса. Нестатические поля объекта класса заполняются конструктором при его создании. Поскольку инициализация поля производится одновременно с его созданием, невозможно создать неинициализированное поле.

Для инициализации полей могут использоваться различные выражения, в том числе, включающие в себя функции. Ниже приведены примеры инициализации полей подобными выражениями:

```
class Demo
{
        static double lg     = Math.Log(10.0);
        string        sz     = "ln(10)";

        static void Main()
        {
                Demo   d      = new Demo();
                Console.WriteLine( "{0} = {1}", d.sz, lg );
        }
}
```

Поскольку статическое поле вызывается в пределах класса, нет необходимости использовать его полное имя.

Наличие явной инициализации поля не отменяет его инициализации значением, используемым по умолчанию. То есть поле сначала инициализируется своим значением, используемым по умолчанию, а затем производится их повторная инициализация указанными значениями. Обе инициализации производятся в порядке определения полей. Такая инициализация за два прохода позволяет создавать циклические выражения инициализации, хотя их применение не только не имеет смысла, но и существенно запутывает программу.

Инициализация полей производится в заголовке класса до того, как будет создан первый объект данного класса. Поэтому при инициализации не могут быть использованы значения нестатических переменных, даже если они были до этого инициализированы.

Статические поля определяются при загрузке класса, а нестатические поля при создании объектов класса. Поэтому статические поля могут использоваться для инициализации нестатических полей даже в том случае, если объявление статического поля расположено за объявлением нестатического поля. Например, следующий фрагмент программы не содержит ошибок:

```
class Demo
{
                int    i      = j;
        static int    j      = k + 100;
        static int    k      = j + 100;

        static void Main()
        {
```

```
            Demo   d      = new Demo();
            Console.WriteLine( "i = {0}, j = {1}, k = {2}", d.i,
Demo.j, Demo.k );
        }
}
```

В результате работы программы будет выведена строка:

```
i = 100, j = 100, k = 200
```

Методы

Методы реализуют функциональные возможности класса. Описание метода состоит из заголовка метода и его тела. Заголовок метода состоит из типа возвращаемого значения, имени метода и круглых скобок, в которых может располагаться список его формальных параметров. Перед типом возвращаемого значения может размещаться набор атрибутов и набор модификаторов метода.

В качестве модификаторов методов могут использоваться следующие ключевые слова:

`new`	`public`
`protected`	`internal`
`private`	`static`
`virtual`	`override`
`abstract`	`extern`

В качестве типа возвращаемого значения может быть указан любой тип или тип `void`.

Метод может принадлежать классу или представлять собой метод одного из реализуемых данным классом интерфейсов. В последнем случае в качестве имени метода используется полное имя метода реализуемого интерфейса, имеющее формат `Interface.Method`, где `Interface` — имя интерфейса, а `Method` — имя метода.

Тело метода представляет собой блок или точку с запятой. Для методов, имеющих модификаторы `abstract` или `extern`, тело метода состоит из точки с запятой. Тело всех остальных методов представляет собой блок, выполняемый при вызове данного метода.

Метод полностью характеризуется своим именем и типом формальных параметров. Имя метода должно отличаться от имен всех членов класса, не являющихся методами. Кроме того, в классе не должно быть методов, различающихся только типом возвращаемого значения.

Поскольку методы являются членами класса, они, так же как и поля могут быть статическими и нестатическими. Статические поля отмечаются модификатором `static`, а нестатические — его отсутствием.

Как уже говорилось выше, статические члены класса можно рассматривать как принадлежащие самому классу, а не его конкретному объекту. Поэтому в статических методах недопустимо использовать ключевое слово `this`. Кроме того, при его объявлении не могут использоваться модификаторы `virtual`, `abstract` и `override`.

Формальные параметры методов

Формальные параметры в списке разделяются запятой. Формальный параметр представляет собой идентификатор типа, перед которым могут стоять набор атрибутов и набор модификаторов аргументов.

В качестве модификаторов аргументов могут использоваться следующие ключевые слова:

`ref`;

`out`;

`params`.

Объявление метода создает отдельную область действия имен для его аргументов и локальных переменных, определенных в блоке, образующем тело данного метода.

При вызове метода создается отдельная копия аргументов и локальных переменных данного метода, после чего производится заполнение копий аргументов их значениями, передаваемыми в списке аргументов. В пределах блока для обращения к аргументам метода производятся по именам, определенным в списке его аргументов.

Существует четыре типа формальных параметров:

1. Передаваемые по значению. Для указания этого типа формальных параметров не требуются никакие модификаторы.
2. Передаваемые по ссылке. Для указания этого типа формальных параметров используется модификатор `ref`.
3. Формальные параметры, используемые для передачи выходных значений. Для указания этого типа формальных параметров используется модификатор `out`.
4. Формальные параметры переменной длины. Для указания этого типа формальных параметров используется модификатор `params`.

Поскольку модификатор формального параметра является частью списка формальных параметров, в классе могут содержаться методы, отличающиеся только модификаторами формальных параметров.

При передаче аргумента по значению в копию данного аргумента, созданную при вызове метода, просто записывается значение, передаваемое в данном аргументе. Копия аргумента рассматривается как локальная перемен-

ная, имеющая имя соответствующего формального параметра, со всеми вытекающими отсюда последствиями. То есть изменение значения этой переменной никак не влияет на значение соответствующего аргумента метода после завершения его работы. Поэтому подобный аргумент может быть создан и инициализирован непосредственно при вызове данного метода. В подобном аргументе могут передаваться константы.

При передаче аргумента по значению значение передаваемого аргумента должно иметь тип данного аргумента или тип, для которого существует явное преобразование к требуемому типу.

Ниже приведен пример передачи аргумента по значению:

```
class Demo
{
        int Fact ( int n )
        {
                int     i = 1;
                for(int j=2; j < n; j++)
                        i       = i*j;
                return i;
        }

        static void Main()
        {
                Demo    d = new Demo();

                Console.WriteLine( d.Fact( 10 ) );
        }
}
```

При передаче аргумента по ссылке в копию данного аргумента, имеющую имя соответствующего формального параметра и созданную при вызове метода, записывается ссылка на данный аргумент. Поэтому все изменения, произведенные с объектом, на который осуществляется ссылка в процессе работы метода, немедленно отражаются на его содержимом.

При вызове метода, передающего один из своих аргументов по ссылке, соответствующий аргумент должен состоять из ключевого слова `ref` и переменной, имеющей тот же тип ссылки, что и формальный параметр. Эта переменная должна быть полностью определена до своего использования в качестве аргумента метода.

Как в самом методе, так и после его вызова, аргумент, передаваемый по ссылке, считается инициализированным.

Ниже приведен пример передачи аргумента по ссылке:

```
class Demo
{
        void Fact ( ref int n )
        {
                int     i = 1;
                for(int j=2; j < n; j++)
                        i       = i*j;

                n = i; }

        static void Main()
        {
                Demo    d = new Demo();
                int             n = 10;

                d.Fact( ref n );

                Console.WriteLine( n );
        }
}
```

Аргумент, в который будет записано выходное значение метода, в основном аналогичен аргументу, передаваемому по ссылке. Основным отличием между ними является то, что для данного аргумента не требуется инициализации до его передачи методу.

При вызове метода аргументы, в которых будут передаваться его возвращаемые значения, должны состоять из ключевого слова `out` и переменной, имеющей тот же тип ссылки, что и формальный параметр. Эта переменная должна быть полностью определена до своего использования в качестве аргумента метода.

В самом методе этот аргумент считается неинициализированным и должен быть инициализирован перед первым обращением к нему.

Аргументы, в которые будут записаны выходные значения метода как правило, используются или для передачи нескольких возвращаемых значений метода, или в том случае, когда возвращаемое значение метода имеет предопределенный тип, например, тип кода ошибки.

Если одним из аргументов метода является одномерный массив произвольного размера, этот аргумент передается последним и имеет модификатор `params`. Поскольку тип массива не определен, это может быть и массив указателей, то есть в данном аргументе могут передаваться массивы `short[]` и `short[][]`, но не могут передаваться массивы `short[,]`.

При вызове методов, имеющих аргумент с модификатором `params`, действует следующее соглашение:

- При передаче в данном аргументе массивов, элементы которого имеют тип, имеющий явное преобразование к типу элементов указанного массива, элементы исходного массива передаются по значению.
- Если вместо данного аргумента при вызове метода стоит выражение, тип которого имеет явное преобразование к типу элемента массива, то этот аргумент и все последующие аргументы, тип которых имеет явное преобразование к типу элемента массива, считаются элементами передаваемого массива. Эти элементы также передаются по значению. При невозможности преобразования типов выдается сообщение об ошибке. При отсутствии аргумента массив имеет нулевую длину.

Ниже приведен пример использования модификаторов `out` и `params`:

```
class Demo
{
        void MiniMax ( out int min, out int max, params int[] a )
        {
                if( a.Length > 0 )
                {
                        min    = a[0];
                        max    = min;

                        for(int i=1; i < a.Length; i++)
                        {
                                if( min < a[i] )
                                        min    = a[i];
                                else
                                        if( max > a[i] )
                                                max    = a[i];
                        }
                }
                else
                {
                        min    = 0;
                        max    = 0;
                }
        }

        static void Main()
        {
```

```
            Demo   d = new Demo();

            int    min, max;

            d.MiniMax( out min, out max, 0, 1, 2, 3 );

            Console.WriteLine("min = {0}, max = {1}", min, max );

            d.MiniMax( out min, out max, new int[] {1, 2, 3, 4} );

            Console.WriteLine("min = {0}, max = {1}", min, max );
        }
}
```

Виртуальные методы

Виртуальные методы отмечаются при своем объявлении модификатором `virtual`. Модификатор `virtual` несовместим с модификаторами `static`, `abstract` и `override`.

В C#, так же как и в C++ разрешена перегрузка и скрытие методов, то есть создание в производных классах методов, имеющих то же самое имя и набор формальных параметров, что и методы базового класса, но выполняющих другие операции.

Если перегружаемый метод не является виртуальным, то при его объявлении метод базового класса скрывается и новый метод вызывается только в методах того класса, в котором он был определен, и в методах классов его потомков, если он сам не будет в них скрыт. В отличие от него виртуальный метод будет вызываться и в методах базовых классов, что позволяет корректировать их работу в соответствии с требованиями производного класса. Какой именно виртуальный метод будет вызван в каждом конкретном случае, определяется в процессе выполнения приложения. Это свойство виртуальных функций называется полиморфизмом, а процесс замены одного метода другим — перегрузкой.

При вызове обычного метода по ссылке на объект класса вызываемый метод определяется, исходя из типа ссылки на объект, а при вызове виртуального метода определяющим является тип самого объекта. Об этом следует помнить при обращении к объекту по ссылке на объект его базового класса. Это свойство сохраняется и для внутренних вызовов данного метода. То есть в методе базового класса будут вызываться виртуальные методы производного класса и обычные методы базового класса.

Предположим, что в процессе выполнения приложения для объекта класса `Derived`, для обращения к которому используется ссылка на объект класса

Base, вызывается метод Method (класс Derived является потомком или совпадает с классом Base). В этом случае вызов метода будет производиться следующим образом:

- производится поиск метода Base.Method среди определенных в данном классе и унаследованных им методов;
- если метод Method не является виртуальным, вызывается найденный метод Base.Method;
- если метод является виртуальным, производится поиск и вызывается метод Derived.Method или его последняя перегрузка в потомках класса Base.

Последней перегрузкой метода Method является:

- объявление в классе Derived метода Method с модификатором virtual;
- объявление в классе Derived метода Method с модификатором override;
- при отсутствии объявления метода Method в классе Derived данные определения применяются к непосредственному базовому классу класса Derived.

Ниже приведен пример использования виртуальных и обычных методов:

```
class Base
{
        public void Simple_Function()
        {
                Console.WriteLine("Base class simple method");
        }
        public virtual void Virtual_Function()
        {
                Console.WriteLine("Base class virtual method");
        }
}
class Derived: Base
{
        new public void Simple_Function()
        {
                Console.WriteLine("Derived class simple method");
        }
        public override void Virtual_Function()
        {
                Console.WriteLine("Derived class virtual method");
        }
}
```

```
class Demo
{
        static void Main()
        {
                Derived        d = new Derived();
                Base           b = d;

                d.Simple_Function();
                b.Simple_Function();
                d.Virtual_Function();
                b.Virtual_Function();
        }
}
```

В результате выполнения данной программы на экране появится текст:

```
Derived class simple method
Base class simple method
Derived class virtual method
Derived class virtual method
```

C# допускает скрытие виртуальных функций. Последствия скрытия виртуальной функции продемонстрированы в следующем примере:

```
class First
{
        public virtual void Some_Func()
        {
                Console.WriteLine("First");
        }
}
class Second: First
{
        public override void Some_Func()
        {
                Console.WriteLine("Second");
        }
}
class Third: Second
{
        new public void Some_Func()
        {
```

```
                Console.WriteLine("Third");
        }
}

class Demo
{
        static void Main()
        {
                Third  t = new Third();
                Second s = t;
                First  f = t;

                f.Some_Func();
                s.Some_Func();
                t.Some_Func();
        }
}
```

В результате выполнения данной программы на экране появится текст:

```
Second
Second
Third
```

На скрывающую функцию не накладывается никаких ограничений. Например, она тоже может быть виртуальной.

Перегрузка методов

В предыдущем разделе уже использовался модификатор `override`. Этим модификатором отмечаются методы, перегружающие виртуальные методы базового класса. Отсутствие этого модификатора в объявлении метода означает, что он скрывает одноименный метод базового класса, имеющий тот же набор формальных параметров. Модификатор `override` несовместим с модификаторами `new`, `static`, `virtual` и `abstract`.

Для любого перегруженного метода производного класса производится поиск перегружаемого им метода по всей иерархии данного класса. Искомый метод должен иметь право доступа `public`, `protected`, `protected internal` или `internal`. В последнем случае базовый класс, в котором располагается перегружаемый метод, должен располагаться в том же проекте, что и производный класс, в котором производится перегрузка.

Перегружаемый метод базового класса должен иметь то же имя и набор формальных параметров, что и перегружающий его метод производного

класса, и модификатор virtual, abstract или override. Перегружаемый метод не может быть статическим или невиртуальным. Перегрузка не должна изменять прав доступа к виртуальному методу.

Для доступа к виртуальному методу базового класса, перегружаемому в данном классе, может быть использовано ключевое слово base. Вызов виртуального метода базового класса с использованием ключевого слова base превращает его в обычный, невиртуальный метод со всеми вытекающими отсюда последствиями. Такая структура, как правило, используется в перегружающем методе для вызова перегружаемого метода, как это показано в приведенном ниже примере:

```
class First
{
        public virtual void Some_Func()
        {
                Console.WriteLine("Class First");
        }
}
class Second: First
{
        public override void Some_Func()
        {
                base.Some_Func();
                Console.WriteLine("Class Second");
        }
}
```

Вызов метода базового класса может быть полезен в том случае, если этот метод выполняет достаточно сложные операции, а изменения, вносимые в него при перегрузке, незначительны и не сказываются на его работе.

Абстрактные методы

Метод называется абстрактным, если при его объявлении был использован модификатор abstract. Абстрактный метод представляет собой виртуальный метод, для которого в данном классе не задана его реализация. Реализация этого метода осуществляется при его перегрузке в производных классах. Поскольку абстрактный метод не содержит реализации, то его тело содержит только точку с запятой. Абстрактные методы могут быть определены только в абстрактных классах. Модификатор abstract несовместим с модификаторами static, virtual и override.

Внешние методы

Если реализация метода расположена за пределами заголовка класса, метод называется внешним. При его объявлении используется модификатор `extern`. Поскольку реализация данного метода в заголовке класса отсутствует, его тело содержит только точку с запятой.

Модификатор `extern`, как правило, используется для определения методов, содержащихся в библиотеках динамической компоновки, поэтому очень часто вместе с ним используется атрибут `DllImport`. Иногда внешние методы используются для получения компактного описания класса. Это особенно актуально в тех случаях, когда тело метода занимает очень много места, что снижает читабельность программ.

Ниже приведен пример создания класса, содержащего функции Win32 API:

```
class Directory
{
     [DllImport("kernel32", setLastError=true)]
     static extern bool CreateDirectory(string Path, SecurityAttributes sa);
     [DllImport("kernel32", setLastError=true)]
     static extern bool RemoveDirectory(string Path);
     [DllImport("kernel32", setLastError=true)]
     static extern int GetCurrentDirectory(int BufSize, StringBuilder Buf);
     [DllImport("kernel32", setLastError=true)]
     static extern bool SetCurrentDirectory(string Path);
}
```

Тело метода

Как уже говорилось выше, тело метода представляет собой блок или точку с запятой. Блок отсутствует в абстрактных и внешних методах.

Если тип возвращаемого значения метода отличен от `void`, в теле метода должен обязательно присутствовать оператор `return`, аргументом которого является выражение, имеющее тип возвращаемого значения метода или имеющее явное преобразование своего типа к типу возвращаемого значения метода, или оператор `throw`, вызывающий исключение. Операторы `return` и `throw` должны быть расположены таким образом, чтобы при любом возможном пути выполнения операций в блоке в нем не мог бы быть достигнут его конец.

Если возвращаемое значение метода имеет тип `void`, то есть данный метод не возвращает значения, в теле метода могут располагаться операторы `return` без аргумента и операторы `throw`, однако, в данном случае при выполнении операций в блоке может быть достигнут его конец.

Все вышесказанное продемонстрировано в следующем примере:

```
class Demo
{
    public    void   First();      // Ошибка. Отсутствует блок
    public    byte   Second() {}  // Ошибка. Должно возвращаться значение
    public    bool   Third( byte b ) // Ошибка. Возможно достижение
                                     // конца блока
    {
              if( b > 0 )
                     return true;
  }
}
```

Свойства

Свойство представляет собой именованный и типизированный атрибут, связанный с объектом или классом. Примером свойства являются длина строки, размер шрифта, заголовок окна, имя пользователя и т. д. Свойства во многом аналогичны полям и имеют одинаковый синтаксис доступа. Однако для хранения свойств не выделяется память. Для доступа к свойству используются специальные процедуры доступа (accessors), определяющие выражение, которое необходимо исполнить при чтении или записи значений свойств. Таким образом, свойства предоставляют возможность чтения и записи атрибутов объекта.

Описание свойства состоит из типа свойства, имени свойства и блока, содержащего процедуры доступа. Перед типом свойства может размещаться набор атрибутов и набор модификаторов свойства.

В качестве модификаторов свойства могут использоваться следующие ключевые слова:

`new`	`public`
`protected`	`internal`
`private`	`static`

Свойство может принадлежать классу или представлять собой непосредственную реализацию одного из свойств интерфейса, реализуемого данным классом. В последнем случае в качестве имени свойства используется полное имя свойства реализуемого интерфейса, имеющее формат `Тип_Интерфейса.Имя_Свойства`.

Хотя свойства имеют тот же синтаксис, что и поля, они не являются переменными и не могут передаваться в аргументах с модификаторами `ref` или `out`.

Статические и нестатические свойства

Если описание свойства включает в себя модификатор `static`, то это свойство является *статическим*. Отсутствие этого модификатора говорит о том, что данное свойство является нестатическим. При описании статического свойства не могут использоваться модификаторы `virtual`, `abstract` и `override`.

Как и любой другой статический член класса, статическое свойство не может принадлежать конкретному объекту класса, и для обращения к нему не может использоваться ключевое слово `this`. Это ключевое слово может использоваться только для обращения к нестатическому методу. Вместо него при обращении к статическому свойству следует использовать имя класса. Таким образом, вызов статического свойства имеет синтаксис `Some_Class.Property`, где `Some_Class` обозначает имя класса, а `Property` — вызываемое свойство, а вызов нестатического метода имеет синтаксис `Some_Object.Property`, где `Some_Object` обозначает имя объекта класса.

Процедуры доступа

Процедуры доступа используются для чтения или изменения состояния свойства. Описание процедуры доступа состоит из ключевого слова `get` или `set` и тела процедуры. Тело процедуры представляет собой блок или точку с запятой. Перед ключевым словом `get` или `set` может стоять модификатор свойства.

Свойство может содержать только одну процедуру доступа для изменения его состояния (с ключевым словом `set`) и одну процедуру доступа для чтения его состояния (с ключевым словом `get`). Порядок расположения описания этих процедур произвольный. Описание любой из этих процедур может отсутствовать, но другая обязательно должна присутствовать.

В качестве модификаторов процедуры доступа могут использоваться следующие ключевые слова:

```
virtual;
override;
abstract;
```

Тело абстрактной процедуры доступа (имеющей модификатор `abstract`) представляет собой точку с запятой. Во всех остальных случаях тело процедуры представляет собой блок.

Процедура доступа для чтения состояния свойства (с ключевым словом `get`) во многом напоминает не имеющий аргументов метод, возвращаемое значение которого соответствует типу свойства, а набор модификаторов представляет собой комбинацию модификаторов свойства и процедуры доступа. При всяком обращении к свойству в программе, исключая случаи присваивания ему нового значения, для данного свойства вызывается процедура чтения

его состояния. Тело данной процедуры должно полностью соответствовать требованиям, предъявляемым к методам, возвращающим значение. Прежде всего это относится к преобразованию типа возвращаемого значения и исключению возможности достичь конца блока. Для исключения этой возможности в процедуре должны использоваться операторы `return` и `throw`.

Процедура доступа для изменения состояния свойства (с ключевым словом `set`) во многом напоминает метод с одним аргументом, не возвращающий значения, набор модификаторов которого представляет собой комбинацию модификаторов свойства и процедуры доступа. Обязательный аргумент процедуры `set` имеет предопределенное имя `value`, тип которого соответствует типу свойства. Использование других имен в данном контексте является ошибкой. Эта процедура вызывается при присваивании свойству нового значения. Тело процедуры изменения состояния свойства должно полностью соответствовать требованиям, предъявляемым к методам, не возвращающим значения. Основным требованием при этом является отсутствие аргумента у оператора `return`.

В зависимости от того, какие процедуры определены для данного свойства, их можно разделить на три группы:

1. Свойства, для которых определены процедуры чтения и изменения состояния называются свойствами *для чтения и записи*.
2. Свойства, для которых определена только процедура чтения, называются свойствами *только для чтения*. Этим свойствам нельзя присваивать новые значения.
3. Свойства, для которых определена только процедура изменения состояния, называются свойствами *только для записи*. Состояние этих свойств нельзя считать или иным образом использовать вне свойства.

Ниже приведен пример объявления процедур доступа:

```
class Demo
{
    private    int    salary;
    public    int    Salary
    {
        get
        {
            return salary;
        }
        set
        {
            if (salary < value)
            {
```

```
                salary = value;
                Console.WriteLine("Congratulations! You've got a raise!");
                }
            }
        }
}
```

В данном примере в классе `Demo` объявлено свойство `Salary`. Процедура доступа `get` этого свойства возвращает размер заработной платы. Процедура `set` проверяет размер новой заработной платы и, если он повысился, изменяет значение переменной `salary` и выводит на экран поздравления. Понижение заработной платы игнорируется как недопустимое.

Ниже приведен пример использования определенного выше свойства:

```
Demo   mySalary      = new Demo();
mySalary.Salary      = 10000;      // Установка значения свойства
Console.WriteLine("My salary is {0}", mySalary.Salary);  // Чтение свойства
```

Набор процедур доступа свойства не является различительным признаком при скрытии свойства в производном классе. То есть при определении в производном классе свойства, имеющего то же имя, что и свойство базового класса, требуется только, чтобы они возвращали значение одного и того же типа, но не требуется, чтобы они имели один и тот же набор процедур доступа. Таким образом, свойство только для чтения может быть скрыто свойством только для записи, что может привести к определенным неудобствам.

Использование свойств позволяет существенно упростить преобразование информации из одной формы представления в другую. Рассмотрим простейший класс, содержащий координату точки в экранных координатах, но позволяющий получать координаты рабочей области окна:

```
class     Point
{
     public     int     x;
     public     int     y;

     public     Point( int x, int y )
     {
          this.x     = x;
          this.y     = y;
     }
}

class Demo
{
```

```
    private     Point     screen;
    private     Point     window;

    public     Demo( Point window )
    {
         this.screen     = window;
         this.window     = window;
    }

    public     Point User
    {
         get
         {
              return new Point( screen.x — window.x, screen.y — window.y );
         }
         set
         {
              screen = new Point( window.x + value.x, window.y + value.y );
         }
    }

    static void Main()
    {
         Demo     d     = new Demo( new Point( 10, 20 )     );

         d.User     = new Point( 30, 40 );

         Console.WriteLine( "X = {0}, Y = {1}", d.User.x, d.User.x );

}
```

В приведенном примере для хранения позиции точки используется класс `Point`, содержащий только конструктор класса и координаты точки. В классе `Demo` хранятся экранные координаты точки, а для работы с пользовательской системой координат используется свойство `User`, в котором производится соответствующее преобразование.

Использование свойств позволяет разделить внутреннюю структуру класса и его внешний интерфейс, что позволяет производить оптимизацию работы класса без внесения изменений в использующие его программы. Это требование предъявлялось еще к классам C++, в которых все переменные должны были быть закрытыми (иметь модификатор права доступа `private`), а

доступ к ним должен был осуществляться через открытые методы (имеющие модификатор права доступа `public`). Однако это требование очень редко выполнялось.

При работе со свойствами не рекомендуется производить в них никаких преобразований, не связанных непосредственно со свойством. Например, если свойство используется для работы с массивом, не рекомендуется производить в нем переход к следующему элементу массива. Это связано не только с тем, что эта операция не всегда необходима и инкремент указателя массива может оказаться "медвежьей услугой" программисту, но и с тем, что при этом возможен выход за границу массива, что потребует специальных средств для обработки данного события.

Свойства могут также использоваться для уменьшения объема оперативной памяти, занимаемой программой. Это достигается, например, за счет того, что в ней не инициализируются неиспользуемые ресурсы. Примером такого похода является фрагмент описания стандартного класса `Console`, используемого нами для вывода информации на экран:

```
public class Console
{
     private static TextReader reader;
     private static TextWriter writer;
     private static TextWriter error;
     public static TextReader In
     {
          get
          {
               if (reader == null)
               {
                    reader = new StreamReader(File.OpenStandardInput());
               }
               return reader;
          }
     }
     public static TextWriter Out
     {
          get
          {
               if (writer == null)
               {
                    writer = new StreamWriter(File.OpenStandardOutput());
               }
```

```
            return writer;
        }
    }
    public static TextWriter Error
    {
        get
        {
            if (error == null)
            {
                error = new StreamWriter(File.OpenStandardError());
            }
            return error;
        }
    }
}
```

Класс `Console` содержит три закрытые переменные `reader`, `writer` и `error`, используемые для работы со стандартными устройствами ввода/вывода и вывода сообщений об ошибках. Кроме того, класс `Console` содержит три свойства: `In`, `Out` и `Error`, используемые для обеспечения доступа к этим переменным. Эти свойства могут использоваться только для чтения.

Как и при любой инициализации переменных, имеющих тип ссылки, переменные `reader`, `writer` и `error` содержат нулевые ссылки. То есть изначально объекты классов для работы с системными устройствами ввода/вывода отсутствуют. При вызове процедуры доступа к свойству в ней проверяется наличие требуемого объекта. Если этот объект отсутствует, он создается. После этого процедура доступа возвращает ссылку на этот объект. При этом используется следующий синтаксис:

```
Console.Error.WriteLine("Unknown error");
```

Виртуальные, перегружаемые и абстрактные процедуры доступа

При объявлении нестатических свойств могут использоваться модификаторы `virtual`, `abstract` и `override`. Наличие модификатора `virtual` или `abstract` или отсутствие модификатора в одной из процедур не накладывает никаких ограничений на соответствующие модификаторы другой процедуры. Однако, если одна из процедур доступа имеет модификатор `override`, то и другая процедура должна иметь тот же модификатор.

Последнее требование связано с тем, что если в свойстве определена перегружаемая процедура доступа, то это свойство считается потомком свойства базового класса и вторая процедура доступа в нем должна или перегружать-

ся, или сохраняться. Если же в свойстве определена неперегружаемая процедура доступа, то это свойство скрывает свойство базового класса, а при скрытии перегрузка невозможна. При скрытии свойства скрываются все его процедуры доступа, то есть, если в свойстве производного класса определена только одна процедура доступа, а в базовом классе была определена отсутствующая процедура, то эта процедура скрывается вместе со свойством базового класса и не переходит в свойство производного класса.

При перегрузке процедур доступа свойство должно иметь то же имя, тот же тип и тот же набор модификаторов права доступа, что и свойство базового класса. Если обе процедуры доступа свойства являются виртуальными, допускается перегрузка как обеих процедур, так и любой из них.

Ниже приведен пример перегрузки и скрытия процедур доступа:

```
abstract class Base_Class
{
        int     i;
        public int First
        {
                virtual get
                {
                        return i + 8;
                }
        }
        public int Second
        {
                get
                {
                        return i;
                }
                virtual set
                {
                        i = value;
                }
        }
        protected int Third
        {
                virtual get
                {
                        return 0;
                }
```

```
                abstract set;
        }
}
class Derived_Class: Base_Class
{
        int j;
        public int First
        {
                set
                {
                        base.i = value - 1;
                }
        }
        public int Second
        {
                override set
                {
                        base.i = value + j;
                }
        }
        protected int Third
        {
                override set
                {
                        j = value;
                }
        }
}
```

В данном примере свойство только для чтения базового класса `First` скрывается в производном классе свойством только для записи. Это может привести к неудобствам при работе с данными классами, но абсолютно корректно. В свойстве `Second` перегружается только процедура `set`, поскольку процедура `get` данного свойства не является виртуальной. Для свойства `Third` перегружается только его абстрактная процедура `set`, в то время как виртуальная процедура `get` осталась без изменений.

События

События позволяют классу посылать сообщения, для обработки которых в приемниках создаются функции обработки событий. Описание события состоит из ключевого слова `event` и объявления поля или свойства. Перед ключевым словом `event` могут располагаться набор атрибутов и набор модификаторов события.

В качестве модификаторов события могут использоваться следующие ключевые слова:

`new`	`public`
`protected`	`internal`
`private`	`static`

Тип события, совпадающий с типом объявляемого поля или свойства, должен представлять собой один из *типов делегатов*, право доступа к которому должно быть не строже, чем к самому событию. Кроме того, в объявлении поля нельзя использовать модификатор `readonly`. Объявление свойства обязательно должно содержать обе процедуры доступа (`get` и `set`), кроме того, для этих процедур не могут использоваться модификаторы `virtual`, `override` или `abstract`.

В классе или структуре, содержащей объявление события, событие рассматривается как закрытое (`private`) поле или свойство соответствующего типа и может использоваться в любом допустимом для него контексте.

Вне этого класса или структуры событие может использоваться только с левой стороны от операций `+=` и `-=`, используемых для добавления функций обработки событий в объект события или удаления из нее этой функции. Контекст использования этого объекта определяется установленным для нее правом доступа.

Такой синтаксис использования объекта события вне его родного класса или структуры позволяет только добавлять или удалять из него функции обработки, но не позволяет получать информацию или воздействовать на его внутреннее состояние.

Для объявления события в классе необходимо, прежде всего, объявить тип делегата для данного события. Несколько событий могут использовать один и тот же тип делегата. При объявлении типа делегата для него указывается список формальных параметров, которые будут передаваться методу, обрабатывающему данное событие. Объявление события аналогично объявлению поля, имеющего тип делегата. Единственное отличие заключается в том, что перед этим объявлением помещается ключевое слово `event`. Как правило, событие является открытым (`public`), но для него могут быть установлены и другие режимы доступа.

Если в классе объявлено событие, то в пределах класса с ним можно работать как с полем, имеющим тип делегата. Если ни один из приемников не

связал делегата с данным событием, это поле имеет нулевое значение. В противном случае это поле содержит ссылку на делегата, который должен вызываться при возникновении данного события. Поэтому при вызове события прежде всего проверяется, не имеет ли оно нулевое значение, и только потом вызывается связанный с ним делегат. Вызов события может производиться только в том классе, в котором оно определено.

Вне класса, в котором оно определено, событие также рассматривается как поле, но при этом существенно ограничивается набор выполняемых с ним действий. Над ним разрешено выполнять только два следующих действия:

- Добавление нового делегата.
- Удаление старого делегата.

Первое действие выполняется операцией +=, а второе действие — операцией –=. Для обработки события приемник создает делегат метода, осуществляющего данную обработку, и подключает этот делегат к событию с использованием операции +=. При этом необходимо, чтобы тип создаваемого делегата совпадал бы с типом события. Для того чтобы приемник прекратил обработку данного типа событий, необходимо удалить из события соответствующий делегат с использованием операции –=.

Ниже приведен пример объявления события:

```
public delegate void EventHandler(object sender, EventArgs e);
public class Demo
{
        public event EventHandler My_Event;
        protected void OnEvent(EventArgs e)
        {
                if (My_Event != null)
                        My_Event(this, e);
        }
        public void Reset()
        {
                My_Event = null;
        }
}
```

Тип делегата `EventHandler` является членом пространства имен `System` и может не указываться. В данном примере член пространства имен `System` скрывается, и вместо него используется полностью эквивалентный ему член текущего пространства имен. На этот тип следует обратить особое внимание, поскольку, хотя стандарт языка C# позволяет событиям использовать любой тип делегата, среда .NET требует, чтобы объявленный тип имел два

формальных параметра, первым из которых является объект, вызывающий событие, а вторым — объект класса, являющегося потомком класса `EventArgs`, имеющий предопределенное имя `e` и содержащий дополнительную информацию о событии. Если нет необходимости передавать какую-либо дополнительную информацию, то второй аргумент имеет тип `EventArgs`, то есть совпадает с предопределенным типом `EventHandler`.

Как уже говорилось выше, в пределах своего класса событие может использоваться в любом контексте. В приведенном примере производится проверка существования события `My_Event`, ему присваивается нулевое значение и для него производится вызов делегата. Для вызова события используется метод `OnEvent`. Вызов события полностью эквивалентен вызову его делегата, поэтому для вызова события не используются никакие специальные языковые конструкции. Перед вызовом делегата необходимо убедиться в его существовании.

Вне класса `Demo` его член `My_Event` может использоваться только в левой части операций `+=` и `-=`. Если определен объект d класса `Demo`, то операция

```
d.My_Event += new EventHandler(...);
```

добавляет новый делегат в список вызова события `My_Event`, а операция

```
d.My_Event -= new EventHandler(...);
```

удаляет делегат из этого списка. Результат этих операций (значение члена `d.My_Event` после их завершения) имеет тип `void`, что предотвращает доступ внешних программ к списку вызова события.

Ниже приведен пример добавления обработчиков событий в класс `Demo`:

```
public class Use_Demo
{
        Demo   First;
        Demo   Second;
        public Use_Demo ()
        {
                First = new Demo();
                First.My_Event += new EventHandler(First_Handler);
                Second = new Demo();
                Second.My_Event += new EventHandler(Second_Handler);
        }
        public void Detach()
        {
                First.My_Event -= new EventHandler(First_Handler);
                First = null;
                Second.My_Event -= new EventHandler(Second_Handler);
```

```
            Second = null;
        }
        void First_Handler(object sender, EventArgs e)
        {
            // Обработка события First.My_Event
        }
        void Second_Handler(object sender, EventArgs e)
        {
            // Обработка события Second.My_Event
        }
}
```

В данном примере конструктор класса подключает обработчики событий к объектам `First` и `Second` класса `Demo`, а метод `Detach` — отключает эти обработчики.

Как правило, события объявляются как поля. Именно так они и были объявлены в приведенном выше примере, однако, в тех случаях, когда такой способ представления событий требует выделения под них значительного объема оперативной памяти, события могут объявляться и как свойства. Такая форма представления событий бывает полезной в тех случаях, когда большинство событий не обрабатывается. Примером таких событий могут быть сообщения об ошибках, число которых во много раз превосходит число ошибок, возникающих в конкретном приложении.

Рассмотрим объявление событий как свойств на примере:

```
using System;
class Demo
{
        // Создание уникальных ключей событий
        static readonly object FirstEventKey = new object();
        static readonly object SecondEventKey = new object();

        // Возвращает дескриптор события, связанный с ключом
        protected Delegate GetEventHandler(object key)
        {
            // Выбор дескриптора события
        }
        // Устанавливает дескриптор события, связанный с ключом
        protected void SetEventHandler(object key, Delegate handler)
        {
            // Установка дескриптора события
        }
```

```
        // Объявление первого события
        public event EventHandler FirstEvent
        {
                get
                {
                   return (EventHandler)GetEventHandler(FirstEventKey);
                }
                set
                {
                        SetEventHandler(FirstEventKey, value);
                }
        }
        // Объявление второго события
        public event EventHandler SecondEvent
        {
                get
                {
                    return (EventHandler)GetEventHandler(SecondEventKey);
        }
                set
                {
                    SetEventHandler(FirstEventKey, value);
                }
        }
}
```

В данном примере метод `SetEventHandler` связывает значение делегата с ключом, а метод `GetEventHandler` возвращает значение делегата, связанное с данным ключом. Поскольку предполагается, что для связи делегата, имеющего значение `null`, с ключом не выделяется оперативной памяти, то необработанные события не занимают оперативную память.

При наследовании событий производными классами возникают некоторые проблемы, связанные с тем, что события могут вызываться только в тех классах, в которых они были определены. Это приводит к тому, что производные классы не могут непосредственно вызывать события, объявленные в их базовом классе, хотя во многих случаях это очень желательно. Для разрешения данного противоречия в базовом классе можно создать защищенный (`protected`) метод, в котором будет вызываться это событие. Еще большей гибкости можно достичь в том случае, если сделать этот метод виртуальным, что позволит производному классу перегружать его. Перегрузка

этого метода позволит производному классу не только вмешаться в процедуру вызова события, объявленного в базовом классе, но и создать для него свою функцию обработки.

Индексаторы

Индексаторы позволяют обращаться к объекту класса как к массиву. Объявление индексатора состоит из типа индексатора, ключевого слова `this`, списка аргументов, располагающегося в квадратных скобках, и объявления процедур доступа, располагающегося в фигурных скобках. Перед типом индексатора может располагаться набор атрибутов и набор модификаторов индексатора. Если индексатор используется для реализации члена интерфейса, перед ключевым словом `this` указывается тип интерфейса, отделяемый от него точкой.

В качестве модификаторов индексатора могут использоваться следующие ключевые слова:

```
new;                    public;
protected;              internal;
private;
```

Элементы списка аргументов отделяются друг от друга запятой. Каждый из этих элементов состоит из типа и идентификатора. Перед типом элемента списка аргументов может размещаться набор атрибутов. Список аргументов не может быть пуст — он должен содержать хотя бы один элемент.

Хотя синтаксис доступа к элементу индексатора полностью аналогичен синтаксису доступа к элементу массива, элемент индексатора нельзя рассматривать как переменную и указывать для него модификаторы `ref` или `out`.

Если класс содержит несколько индексаторов, они должны отличаться списком аргументов. При этом учитывается только число и тип аргументов, но не их имена.

Индексаторы во многом аналогичны свойствам, но между ними существуют следующие различия:

- свойства различаются по своему имени, а индексаторы — по списку аргументов;
- для доступа к свойству используется его имя или его полное имя (с указанием имени класса или объекта класса), к индексатору можно обратиться только для получения его элемента;
- свойство может быть статическим, а индексатор — нет;
- процедура `get` свойства аналогична методу без списка формальных параметров, а та же самая процедура доступа индексатора имеет список формальных параметров, совпадающий со списком аргументов индексатора;

- процедура `set` свойства аналогична методу с одним формальным параметром, а та же самая процедура доступа индексатора имеет список формальных параметров, совпадающий со списком аргументов индексатора и дополнительный аргумент `value`;
- в процедуре доступа индексатора не может быть объявлена локальная переменная, имя которой совпадает с именем аргумента индексатора.

Ниже приведен пример использования индексатора:

```
class  Hex_Array
{
       int[]  buffer;
       int    length;
       public Hex_Array(int length)
       {
              if (length < 0)
                     throw new ArgumentException();
                     buffer = new int[((length – 1) >> 2) + 1];
                     this.length = length;
       }
       public int Length
       {
              get
              {
                     return length;
              }
       }

       public byte this[int index]
       {
              get
              {
                     if (index < 0 || index >= length)
                     {
                            throw new IndexOutOfRangeException();
                     }
        return (byte) ((buffer[index >> 2] >> ((index & 3) << 2)) & 15);
              }
              set
              {
                     if (index < 0 || index >= length)
                     {
```

```
                        throw new IndexOutOfRangeException();
                }
                int    i = (index & 3) << 2;
                int    j = index >> 2;
                buffer[j] &= ~(15 << i);
                buffer[j] |= (value & 15) << i;

        }
    }

    static void Main()
    {
        int    n;

        Hex_Array     h      = new Hex_Array( 32  );

        h[5]   = 5;
        h[7]   = 7;
        h[4]   = 4;
        h[8]   = 8;
        h[6]   = 6;
        h[10]  = 10;
        h[9]   = 9;
        h[11]  = 11;

        for(n=4; n < 12; n++ )
        {
            Console.WriteLine( "h[{0}] = {0}", n );
        }

        for(int i=0; i < 65636; i++)
        {
            for(int j=0; j < 8192; j++)
            {
                n = 0;
            }
        }
    }
}
```

В приведенном выше примере индексатор используется для доступа к целочисленному массиву как к массиву четырехразрядных переменных. Для проверки правильности работы индексатора выводимый на экран массив заполняется в случайном порядке.

Операции

В C#, так же как и в C++ имеется возможность определить *операции*, которые могут использоваться для объектов класса. Различают исполнительные операции и операции преобразования типов. Объявление исполнительной операции состоит из списка модификаторов, типа возвращаемого значения, ключевого слова `operator`, идентификатора операции, списка аргументов и блока. Объявление операции преобразования типа состоит из списка модификаторов, ключевого слова `implicit` или `explicit`, ключевого слова `operator`, возвращаемого типа, списка аргументов и блока.

Ниже приведены основные требования, предъявляемые ко всем операциям:

- набор модификаторов операции должен содержать модификаторы `public` и `static`, расположенные в любой последовательности. Использование других модификаторов или отсутствие любого из указанных модификаторов недопустимо;
- аргументы операции должны передаваться только по значению. Недопустимо использовать для них модификаторы `ref` или `out`;
- все типы ссылок, используемые в операции, должны иметь право доступа не жестче, чем сама операция.

Как и любые другие методы, операции наследуются производными классами, однако, они являются статическими методами и, следовательно, не могут быть виртуальными, о чем можно только сожалеть. Кроме того, операцию нельзя скрыть. Это связано с тем, что в объявлении операции обязательно присутствует имя класса или структуры, в которой она объявлена. Это исключает использование для них модификатора `new`.

Унарные операции

Унарные операции имеют один аргумент, имеющий тип класса или структуры, в котором они определены. К унарным операциям относятся `+`, `–`, `!`, `~`, `++`, `--`, `true` и `false`. При этом операции `+`, `–`, `!` и `~` могут возвращать значение любого типа, операции `++` и `--` должны возвращать значение того же типа, что и их аргумент, а операции `true` и `false` должны возвращать значение типа bool.

Операции `true` и `false` можно определять только совместно, поскольку они используются для преобразования типа класса или структуры, в которой они определены, к типу логического значения. Это требует однозначного определения обоих альтернатив. При определении только одной из этих операций транслятор выдает сообщение об ошибке.

Бинарные операции

Бинарные операции имеют два аргумента, по крайней мере один из которых должен иметь тип класса или структуры, в которой определена данная операция. К бинарным операциям относятся `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=` и `<=`.

Некоторые из этих операции могут определяться только совместно. Иными словами, или не должно быть определено ни одной операции из этой пары, или должны быть определены обе операции. При этом эти операции должны иметь одинаковый набор формальных параметров и тот же тип возвращаемого значения. Ниже приведен список совместно определяемых операций:

- `operator ==` и `operator !=`;
- `operator >` и `operator <`;
- `operator >=` и `operator <=`.

Операции преобразования типов

Операции преобразования типов используются для явного или неявного преобразования типа класса или структуры к другому типу. Для задания неявного преобразования типов используется ключевое слово `implicit`. Неявные преобразования производятся в том случае, когда не задано явное преобразование, а требуемый тип ссылки или аргумента функции отличается от типа объекта. Для задания явного преобразования типов используется ключевое слово `explicit`. Явное преобразование типов используется при обращении к объекту по ссылке в тех случаях, когда отсутствует неявное преобразование или необходимо разрешить возникшую неоднозначность при выборе типа, к которому производится преобразование.

Данная операция преобразует исходный тип, соответствующий типу аргумента операции, в преобразованный тип, соответствующий типу его возвращаемого значения. При этом:

- исходный и преобразованный типы не должны совпадать;
- один из этих типов должен быть типом класса или структуры, в которой определена данная операция;
- ни один из этих типов не может быть типом интерфейса или типом `object`;
- исходный тип не может быть ни базовым, ни производным типом для преобразованного типа.

Отсюда следует, что операция преобразования типов может только преобразовывать другие типы в тип своего класса или структуры или же преобразовывать этот тип в другие типы. Последнее правило и запрещение преобразования из типа `object` и в него, что является частным случаем этого правила, связано с тем, что операции преобразования типов не могут пере-

определять предопределенные операции преобразования, создаваемые в том числе и при наследовании.

Запрет на преобразование в типы интерфейсов и из них гарантирует невозможность доступа к интерфейсу, не реализованному ни в одном классе.

Операция преобразования типов является единственным исключением из правила о том, что два члена класса не могут различаться только своим возвращаемым значением. Таким образом, эти операции различаются типом возвращаемого значения и типом аргумента. Ключевые слова `implicit` и `explicit` не упомянуты среди различающих признаков, поэтому нельзя создать версии для явного и неявного преобразования одной и той же пары типов.

Как правило, при неявном преобразовании типов не должны вызываться исключения или теряться какая-либо информация. Если при преобразовании может возникнуть необходимость вызова исключения (например, вследствие переполнения) или может быть потеряна информация (преобразованный тип имеет меньшую разрядность, чем исходный тип), необходимо использовать явное преобразование, как это показано в приведенном ниже примере:

```
public struct integer
{
        short  value;
        public integer(short value)
        {
                this.value = value;
        }
        public static implicit operator int(integer i)
        {
                return i.value;
        }
        public static explicit operator integer(int i)
        {
                if (i < -32768 || i > 32767)
                        throw new ArgumentException();
                return new integer((short) i);
        }
}
```

Обратите внимание на то, что при преобразовании типа `int` в тип `integer` не только производится преобразование типа, но и изменяется значение поля данной структуры. Если значение преобразуемой целочисленной величины превышает диапазон значений этого поля, в операции преобразования вызывается исключение.

Конструкторы объектов класса

Эти *конструкторы* используются для инициализации объектов класса. Объявление конструктора состоит из имени конструктора, совпадающего с именем класса, за которым следуют круглые скобки и блок. В круглых скобках может располагаться список формальных параметров конструктора, а за ними может стоять инициализатор конструктора. Инициализатор конструктора состоит из двоеточия, ключевого слова `base` или `this` и круглых скобок, в которых может располагаться список аргументов. Перед именем конструктора может располагаться набор атрибутов и набор модификаторов конструктора.

В качестве модификаторов индексатора могут использоваться следующие ключевые слова:

`public`	`protected`
`internal`	`private`

Конструктор класса можно рассматривать как особый метод, используемый для инициализации членов класса, поэтому требования к списку формальных параметров конструктора полностью совпадают с требованиями к списку формальных параметров метода. В том числе продолжает действовать требование, что режим доступа к формальному параметру конструктора класса, имеющему тип ссылки, не может быть более жестким, чем к самому конструктору класса. Поскольку имя конструктора класса не может быть изменено, различные конструкторы одного и того же класса должны различаться списком своих формальных параметров.

Инициализатор конструктора определяет конструктор, который должен быть вызван до того, как управление будет передано первому оператору блока данного конструктора класса. Этот способ вызова конструктора, заимствованный из C++, имеет один существенный недостаток. Дело в том, что вызываемому в инициализаторе конструктору необходимо передать аргументы. Эти аргументы берутся из списка формальных параметров вызываемого конструктора. В некоторых случаях формальные параметры этих конструкторов не совпадают и требуют проведения вычислений для их преобразования. При таком способе вызова конструктора все эти вычисления приходится производить в инициализаторе.

Использование инициализатора конструктора связано с тем, что конструкторы класса не наследуются, однако, при создании объекта класса необходимо проинициализировать все унаследованные им члены. Для этого в инициализаторе конструктора и вызывается конструктор базового класса.

Блок конструктора содержит операторы инициализации членов класса, которые не были проинициализированы в статическом конструкторе. Конструктор класса не возвращает значения, поэтому к его блоку предъявляются те же требования, что и к блоку соответствующего метода.

Если в классе не указан конструктор, то при создании его объекта вызывается конструктор, используемый по умолчанию.

Инициализатор конструктора

Как уже говорилось выше, конструктор любого класса, кроме класса `object`, должен произвести инициализацию всех унаследованных им членов. Принципы объектно-ориентированного программирования требуют, чтобы все операции над переменными производились по возможности методами объектов, в которых они определены. Это касается и их инициализации. Поэтому, прежде чем приступить к инициализации членов своего класса этот конструктор должен вызвать конструктор базового класса, инициализирующий свои члены. Если этот базовый класс не является классом `object`, его конструктор, прежде чем приступить к работе, вызовет конструктор своего базового класса. Таким образом, обеспечивается инициализация всех членов класса (в том числе и скрытых).

Если в конструкторе класса не указан инициализатор, то к нему автоматически добавляется инициализатор `base()`.

Различают два типа инициализаторов:

- Инициализатор `base(...)` сразу вызывает конструктор базового класса. При этом среди всех определенных конструкторов базового класса выбирается тот, который полностью соответствует правилам, применяемым при перегрузке методов. Если в базовом классе не может быть найден подходящий конструктор, транслятор выдает сообщение об ошибке.
- Инициализатор `this(...)` вызывает другой конструктор того же класса. При этом среди всех конструкторов данного класса выбирается тот, который полностью соответствует правилам, применяемым при перегрузке методов. Если в данном классе не может быть найден подходящий конструктор, транслятор выдает сообщение об ошибке. Сообщение об ошибке возникает и в том случае, если выбирается тот конструктор класса, в котором присутствует данный инициализатор. Использование подобного инициализатора является большим шагом вперед по сравнению с C++, в котором вызов одного конструктора класса из другого конструктора был недопустим. Это приводило к необходимости написания нескольких практически идентичных конструкторов, различающихся, как правило, только набором формальных параметров и несколькими простейшими преобразованиями полученных аргументов.

Для обеспечения возможности передачи аргументов конструктора класса конструкторам, вызываемым в его инициализаторе, инициализатор включен в область действия формальных параметров конструктора. При этом следует помнить о том, что на момент вызова инициализатора объект еще не создан, поэтому в нем можно использовать только статические члены класса.

Инициализация полей класса

Если в конструкторе класса не указан инициализатор или указан инициализатор вида `base( ... )`, то после вызова соответствующего конструктора и перед передачей управления первому оператору блока производится инициализация полей класса.

Поскольку каждое поле является объектом, оно имеет свой конструктор, используемый по умолчанию. Как правило, это конструктор заполняет все биты данного поля нулевыми значениями. Эти конструкторы и вызываются для первичной инициализации полей. После этого производится инициализация полей значениями, указанными при их объявлении. Инициализация полей производится в порядке их объявления в классе.

Рассмотрим следующий пример:

```
abstract class Base_Class
{
        public Base_Class()
        {
                Test_Print();
        }
        public abstract void Test_Print();
}
class Derived_Class: Base_Class
{
        int     i       = 1;
        int     j;
        public Derived_Class() {}
        public Derived_Class(int n): this()
        {
                j       = n;
        }
        public override void Test_Print()
        {
                Console.WriteLine("i = {0}, j = {1}", i, j);
        }
        static void Main()
        {
                Derived_Class  b       = new Derived_Class( 10 );
        }
}
```

При создании объекта класса `Derived_Class` будет вызван второй конструктор данного класса, поскольку при создании объекта задан параметр. При этом, прежде всего, будут проинициализированы поля данного класса. Поле `i` будет проинициализировано дважды: после первой инициализации он будет содержать нулевое значение, а после второй — единичное значение. Поле `j` будет проинициализировано только один раз и будет содержать нулевое

значение. После этого будет вызван первый конструктор данного класса. Поскольку он не содержит инициализатора, то будет вызван конструктор базового класса без параметров. В этом конструкторе будет вызвана перегруженная виртуальная функция, выводящая значения полей производного класса. При этом на экран будет выведена строка

```
i = 1, j = 0
```

После вывода этой строки на экран будет выполнен блок первого конструктора производного класса, не содержащий операторов. Затем будет выполнен блок второго конструктора производного класса, и в поле `j` будет записано значение `10`, переданное в аргументе конструктора.

Статические поля класса могут использоваться до его объявления. При этом используются значения, устанавливаемые в них конструкторами, используемыми по умолчанию. После объявления класса эти поля содержат значения, установленные при их инициализации в классе. Это позволяет использовать циклическое определение статических переменных, как это продемонстрировано в следующем примере:

```
class First
{
        public static int i = Second.i + 1;
}
class Second
{
        public static int i = First.i + 1;
        static void Main()
        {
                Console.WriteLine("First = {0}, Second = {1}", First.i,
Second.i);
        }
}
```

Хотя оба класса имеют аналогичную инициализацию своих статических полей, в результате выполнения этой программы на экране появится строка

```
First = 1, Second = 2
```

Различие значений объясняется тем, что поскольку класс `First` был объявлен раньше, при инициализации его поля использовалось нулевое значение поля класса `Second`, а при инициализации поля класса `Second` — единичное значение поля класса `First`.

Однако если метод `Main` будет перенесен в класс `First`, на экран будет выведена строка

```
First = 2, Second = 1
```

Данная программа работает корректно, поскольку класс, содержащий метод `Main`, инициализируется последним. Однако исходя из того, что в более сложной программе последовательность инициализации классов не может быть так просто определена, следует, по возможности, избегать циклических инициализаций полей.

Конструкторы класса, используемые по умолчанию

Если в классе не определен конструктор, то в него по умолчанию добавляется конструктор

```
public Class_Name(): base() {}
```

где `Class_Name` — имя класса. Этот конструктор производит инициализацию полей и вызывает конструктор без параметров базового класса. Если в базовом классе отсутствует или недоступен конструктор без параметров, транслятор выдает сообщение об ошибке. Таким образом, определение в классе любого конструктора исключает из него конструктор, используемый по умолчанию.

Закрытые конструкторы классов

Если класс содержит только закрытые конструкторы, то ни один другой класс, кроме вложенных в него классов, не может создать объект данного класса. Обычно закрытые конструкторы используются для запрета доступа к конструктору, используемому по умолчанию в классах, содержащих только статические члены. Например:

```
public class Metric
{
        private Metric() {}   // Предотвращает вызов конструктора,
                              // используемого по умолчанию
        public const double inch = 2.54;
        public static double In2Cm(double x) { x*inch }
        public static double Cm2In(double x) { x/inch }
}
```

Данный пример содержит класс, производящий преобразование из дюймов в сантиметры и обратно.

Деструкторы класса

Деструктор класса используется для корректного завершения работы объекта класса и освобождения выделенной ему памяти. Объявление деструктора класса состоит из идентификатора, состоящего из символа ~ и имени класса, пары круглых скобок и блока. Перед идентификатором деструктора конструктора может располагаться набор атрибутов.

Блок деструктора содержит операторы, выполняющие все необходимые действия для корректного завершения работы объекта класса и освобождения занимаемой им оперативной памяти. Поскольку деструктор класса не возвращает значения, поэтому к его блоку предъявляются те же требования, что и к блоку метода, не возвращающего значения.

Деструкторы не наследуются, и в классе может быть определен только один деструктор. По завершении своей работы деструктор автоматически вызывает деструктор базового класса. Вызов деструктора для объекта возможен только после того, как объект класса становится недоступным для всех классов приложения. Однако это не означает, что он будет немедленно вызван. В связи с тем, что явный вызов деструктора запрещен, это может привести к существенному перерасходу оперативной памяти, тем более что многие объекты могли бы быть уничтожены задолго до того, как они выйдут за пределы своей области определения. Это накладывает существенные ограничения на область применения языка C#, не позволяя использовать его в серьезных приложениях, работающих в системах с ограниченными ресурсами, например, на различных серверах.

Как уже говорилось в главе 3, деструкторы класса C# существенно отличаются от деструкторов классов C++. Это отличие заключается, прежде всего, в том, что в классах C# отсутствуют деструкторы, используемые по умолчанию, и объявление в классе деструктора существенно затрудняет работу сборщика мусора, поскольку в этом случае класс уничтожается за два его прохода. После первого вызова сборщика мусора для объекта, имеющего деструктор и завершившего свою работу, в потоке завершения вызывается его деструктор, и ссылка на него удаляется из списка объектов, требующих завершения. Память, занимаемая объектом, освобождается только после второго вызова сборщика мусора. Если бы объект не имел деструктора, то память была бы освобождена после первого же вызова сборщика мусора (если бы он просмотрел этот объект).

См. главу 3.

Поэтому следует ясно представлять себе недостатки включения деструктора в класс:

- для данных объектов задействованы дополнительные ресурсы и, кроме того, усложнена и замедлена процедура их удаления;
- завершение работы объекта выполняется в отдельном потоке;
- порядок завершения объектов не определен, поэтому могут возникнуть конфликты при вызове в деструкторе объектов, также имеющих деструктор;
- при завершении работы приложения деструкторы не вызываются. Их можно вызвать принудительно, применив метод `System.GC.RequestFinalizeOnShutdown`, но это не рекомендуется, поскольку замедляет процесс закрытия приложения.

Если необходимость в вызове деструктора класса отпала, его вызов можно отменить, вызвав метод `System.GC.SupressFinalize`.

Статические конструкторы класса

Статический конструктор осуществляет инициализацию статических членов класса. Иногда говорят, что он инициализирует сам класс до инициализации его объектов. Объявление статического конструктора класса состоит из ключевого слова `static`, идентификатора, совпадающего с именем класса, пары круглых скобок и блока. Перед идентификатором деструктора конструктора может располагаться набор атрибутов.

Блок статического конструктора содержит операторы инициализации статических членов класса, Поскольку конструктор класса не возвращает значения, к его блоку предъявляются те же требования, что и к блоку соответствующего метода.

Статические конструкторы, как и любые другие конструкторы, не наследуются. Они вызываются только автоматически и не могут быть вызваны в явном виде. Хотя время и порядок вызова статических конструкторов не определен, можно с уверенностью рассчитывать на то, что:

- статический конструктор класса будет вызван до создания первого объекта данного класса;
- статический конструктор класса будет вызван до первой ссылки на любой статический член данного класса;
- статический конструктор класса будет вызван до вызова статического конструктора любого его производного класса;
- статический конструктор класса выполняется не более одного раза.

Например, в процессе работы следующей программы

```
using System;
class Demo
{
        static void Main()
        {
                First.Some_Function();
                Second.Some_Function();
        }
}
class First
{
        static First()
        {
                Console.WriteLine("First Initialization");
        }
```

```
        public static void Some_Function()
        {
                Console.WriteLine("First Function");
        }
}
class Second
{
        static Second()
        {
                Console.WriteLine("Second Initialization");
        }
        public static void Some_Function()
        {
                Console.WriteLine("Second Function");
        }
}
```

на экран будет выведен текст

```
First Initialization
Second Initialization
First Function
Second Function
```

или текст

```
First Initialization
First Function
Second Initialization
Second Function
```

Если же в этом примере класс `First` сделать потомком класса `Second`, то на экран всегда будет выводиться текст

```
Second Initialization
First Initialization
First Function
Second Function
```

Глава 11

Интерфейсы

Интерфейсы представляют собой абстрактный тип ссылки, реализуемый в классах приложения. Они во многом аналогичны абстрактным классам, все члены которых являются абстрактными.

Заголовок интерфейса

Заголовок класса состоит из ключевого слова `interface`, идентификатора интерфейса и тела интерфейса. Перед ключевым словом `interface` могут располагаться набор атрибутов и набор модификаторов интерфейса. После идентификатора интерфейса может находиться спецификатор базового интерфейса. За телом интерфейса, представляющим собой блок, может стоять точка с запятой.

Модификаторы интерфейса

В качестве модификаторов интерфейса могут выступать ключевые слова:

- `new`
- `public`
- `protected`
- `internal`
- `private`

Каждый из этих модификаторов может появляться в заголовке идентификатора только один раз. Модификатор `new` может использоваться только для вложенного интерфейса. Он подавляет предупреждение, выдаваемое при скрытии члена базового интерфейса.

Модификаторы `public`, `protected`, `private` и `internal` определяют режим доступа к интерфейсу.

Спецификатор базового интерфейса

Язык программирования C# поддерживает множественное наследование интерфейсов, то есть каждый интерфейс может иметь любое количество базовых интерфейсов или вообще не иметь базового интерфейса. Эти интерфейсы называются *явными базовыми интерфейсами*. Список явных базовых интерфейсов отделяется от идентификатора интерфейса двоеточием, а элементы списка отделяются друг от друга запятой.

Производные интерфейсы не должны расширять права доступа к своим явным базовым интерфейсам. Например, интерфейс, имеющий право доступа `public`, не может иметь явного базового интерфейса с правом доступа `private` или `protected`.

Базовыми интерфейсами являются явные базовые интерфейсы и явные базовые интерфейсы их базовых интерфейсов. Ни один интерфейс не может быть собственным базовым интерфейсом.

Интерфейс наследует все члены своих базовых интерфейсов. Таким образом, класс или структура, реализующая интерфейс, неявным образом реализует все методы его базовых интерфейсов.

Члены интерфейса

Членами интерфейса являются члены всех его базовых классов и определенные в нем члены. В интерфейсе не может быть не определено ни одного члена. Членами интерфейса могут быть методы, свойства, события и индексаторы. Интерфейс не может содержать констант, полей, операторов, конструкторов, деструкторов, статических конструкторов, определений типов и любых статических членов.

По умолчанию все интерфейсы являются открытыми (имеют модификатор права доступа `public`). Это право доступа распространяется на все члены класса, которые не могут иметь никаких модификаторов, кроме модификатора `new`.

Ниже приведен пример интерфейса, содержащего все возможные типы членов:

```
public delegate void DemoEvent(IDemo sender);
public interface IDemo
{
        void    Some_Func( int i);
        int     Some_Property { set; }
        event   DemoEvent Some_Event;
        int     this[int index] { get; set; }
}
```

Заголовок интерфейса создает новую область видимости имен для своих членов. При объявлении членов интерфейса действуют следующие правила:

- имя метода должно отличаться от имен всех свойств и событий, объявленных в том же интерфейсе. Кроме того, методы с одинаковыми именами должны различаться числом и типом своих формальных параметров;
- имена свойств и событий должны отличаться от имен всех членов, объявленных в данном интерфейсе;
- в интерфейсе не могут быть определены два индексатора с одинаковым набором формальных параметров.

Унаследованные члены интерфейса не подчиняются сформулированным выше правилам. Поэтому в интерфейсе может быть определен член с тем же именем и набором формальных параметров, что и в базовом интерфейсе. Поскольку перегрузка членов интерфейса запрещена, здесь речь может идти только о скрытии члена базового интерфейса. Как и в любом другом случае, скрытие не является ошибкой, но при его обнаружении транслятор выдает предупреждение. Для подавления этого предупреждения используется модификатор `new`. Если же этот модификатор устанавливается при объявлении члена интерфейса, не скрывающего член базового класса, то, обнаружив это, транслятор снова выдает предупреждение.

Методы

Описание метода интерфейса состоит из типа возвращаемого значения, идентификатора метода, пары круглых скобок, в которых может располагаться список формальных параметров, и тела метода. Перед типом возвращаемого значения может стоять набор атрибутов и модификатор `new`. Поскольку интерфейс представляет собой абстрактный тип, тело метода интерфейса всегда представляет собой точку с запятой. Метод интерфейса аналогичен абстрактному методу класса.

Свойства

Описание свойства состоит из типа свойства, имени свойства и блока, содержащего процедуры доступа. Перед типом свойства может размещаться набор атрибутов и модификатор `new`. Поскольку интерфейс представляет собой абстрактный тип, тело процедур доступа всегда представляет собой точку с запятой. Свойство интерфейса аналогично свойству класса за тем исключением, что в нем определены абстрактные процедуры доступа.

События

Описание события состоит из ключевого слова `event`, типа события и его идентификатора, за которым следует точка с запятой. Перед ключевым словом `event` может размещаться набор атрибутов и модификатор `new`. Поскольку интерфейс представляет собой абстрактный тип, тело процедур доступа всегда представляет собой точку с запятой. Событие интерфейса аналогично событию класса.

Индексаторы

Объявление индексатора состоит из типа индексатора, ключевого слова `this`, списка аргументов, располагающегося в квадратных скобках, и объявления процедур доступа, располагающегося в фигурных скобках. Перед типом индексатора может размещаться набор атрибутов и модификатор `new`. Поскольку интерфейс представляет собой абстрактный тип, тело процедур доступа всегда представляет собой точку с запятой.

Доступ к членам интерфейса

Доступ к членам интерфейса всегда должен производиться по их полному имени, с указанием имени интерфейса или через индексатор. Если определен интерфейс `Interface`, то доступ ко всем его членам, кроме индексатора, можно получить, используя синтаксис `Interface.Member`, где `Member` — имя члена интерфейса, а к индексатору — используя синтаксис `Interface[Arguments]`, где `Arguments` — набор аргументов, соответствующих списку формальных параметров одного из индексаторов интерфейса.

Если при создании интерфейса не использовалось множественное наследование, то есть все базовые интерфейсы данного интерфейса имели не более одного явного базового интерфейса, то правила просмотра полей, вызова методов и доступа по индексу не отличаются от правил, действующих для классов и структур: члены производного класса, имеющие то же имя и набор формальных параметров, что и члены базового класса, скрывают соответствующие члены базового класса.

При множественном наследовании могут возникать конфликты, если в нескольких не связанных между собой базовых классах будут определены несколько членов, имеющих одно и тоже имя и набор формальных параметров. Для разрешения этой ситуации необходимо произвести явное преобразование к типу требуемого интерфейса.

Рассмотрим пример:

```
interface IFirst
{
        int Something { get; set; }
}
interface ISecond
{
        void Something(int i);
}
interface IThird: IFirst, ISecond   {}
class Demo
{
        void Some_Func(IThird i)
        {
                int    k;
                k = i.Something;             // Ошибка. Хотя понять можно
                i.Something(k);              // Ошибка. Хотя понять можно
                k = ((IFirst)i).Something;   // Вызов IFirst.Something.get
```

```
            ((ISecond)x).Something(k); // Вызов метода ISecond.Something
        }
}
```

Первые два обращения к интерфейсу содержат ошибки, поскольку не позволяют распознать, к члену какого именно класса осуществляется доступ, хотя вызов метода и обращение к свойству спутать сложно. В двух следующих обращениях произведено необходимое преобразование типов и неоднозначность разрешена.

Рассмотрим пример скрытия членов базового класса:

```
interface IBase
{
        void Some_Func(byte b);
}
interface IFirst: IBase
{
        new void Some_Func(byte b);
}
interface ISecond: IBase
{
        void Other_Func( int i);
}
interface IThird: IFirst, ISecond {}
class Demo
{
        void Test(IThird t)
        {
                t.Some_Func(7);              // Вызов IFirst.Some_Func
                ((IBase)t).Some_Func(3);     // Вызов IBase.Some_Func
                ((IFirst)t).Some_Func(1);    // Вызов IFirst.Some_Func
                ((ISecond)t).Some_Func(0);   // Вызов IBase.Some_Func
        }
}
```

Поскольку существуют два пути доступа к функции `Some_Func`, а она скрывается только в одном из них, в приведенном примере присутствует неоднозначность при ее вызове. В этом случае при множественном наследовании используется простое правило скрытия функций: если функция скрывается в одном из путей доступа к ней, она скрывается для производного класса.

Полные имена членов интерфейса

Для более точного указания, какому именно интерфейсу принадлежит конкретный член, используются *полные имена*. Полное имя члена интерфейса состоит из имени интерфейса, в котором определен данный член, точки и имени члена интерфейса. Полные имена функций использовались в комментариях к примерам предыдущего раздела.

Если интерфейс определен в некотором пространстве имен, то полное имя его члена состоит из имени пространства имен, имени интерфейса и имени члена интерфейса, разделенных точками. Так полное имя метода `Some_Func` в следующем примере будет `Demo.IBase.Some_Func`.

```
namespace Demo
{
        interface IBase
        {
                void Some_Func(byte b);
        }
}
```

Поскольку интерфейсы не могут быть статическими, доступ к членам интерфейса по их полному имени возможен только при реализации интерфейсов.

Реализация интерфейсов

Поскольку интерфейсы являются абстрактными типами, их члены нуждаются в реализации, которая может осуществляться в классах и структурах. Для указания того, что некоторый интерфейс реализуется в данном классе или структуре, его имя включается в список базовых интерфейсов. Список базовых интерфейсов класса помещается за именем его базового класса и отделяется от него запятой. Если имя базового класса не указано, запятая не ставится.

Ниже приведен пример реализации интерфейса в классе:

```
interface IFirst
{
        double First_Func( double i );
}
interface ISecond
{
        byte   Second_Func( byte b);
}
class Demo: IFirst, ISecond
{
```

```
        public double First_Func( double i )
        {
                // Реализация функции
        }
        public byte   Second_Func( byte b)
        {
                // Реализация функции
        }
}
```

Реализацию методов и процедур доступа интерфейса можно было бы рассматривать как перегрузку. Поскольку все члены интерфейса являются абстрактными, при перегрузке их членов не указывается модификатор `override`.

Явная реализация членов интерфейса

При реализации интерфейса в классе или структуре может быть использована *явная реализация членов интерфейса*. При явной реализации члена интерфейса используется его полное имя. Например:

```
interface IFirst
{
        double First_Func( double i );
}
interface ISecond
{
        byte   Second_Func( byte b);
}
class Demo: IFirst, ISecond
{
        double IFirst.First_Func( double i )
        {
                // Реализация функции
        }
        byte   ISecond.Second_Func( byte b)
        {
                // Реализация функции
        }
}
```

В отличие от предыдущего примера здесь произведена явная реализация членов интерфейса. Явная реализация членов интерфейса позволяет реализовывать не только его члены, но и скрытые члены его базовых классов.

Явная реализация члена интерфейса не может содержать модификаторов права доступа и модификаторов `abstract`, `virtual`, `override` и `static`.

Режим доступа к членам интерфейса, имеющим явную реализацию, отличается от режима доступа ко всем остальным членам класса или структуры. С одной стороны, поскольку вызов метода или доступ к свойству интерфейса, имеющему явную реализацию, невозможен по их полному имени, их можно рассматривать как закрытые (`private`) члены класса. С другой стороны, поскольку к ним возможен доступ через объект интерфейса, они могут рассматриваться как открытые (`public`) члены класса.

Явная реализация членов интерфейса используется, как правило, для решения одной из двух задач:

1. Предотвращения непосредственного доступа к интерфейсу со стороны пользователей класса или структуры. Это бывает полезным в том случае, если данный класс или структура являются элементом пользовательского интерфейса, используемого для доступа к системным ресурсам.
2. Разрешения конфликтов, возникающих в тех случаях, когда различные базовые интерфейсы содержат члены, имеющие одинаковые имена и списки формальных параметров. Этот конфликт становится еще жестче, если при этом члены интерфейса имеют различные типы возвращаемых значений.

При явной реализации члена интерфейса в список базовых интерфейсов необходимо включить интерфейс, в котором был определен данный метод, или один из его потомков. При этом в полном имени члена интерфейса следует указывать тот интерфейс, в котором был определен перегружаемый метод. Это продемонстрировано в приведенном ниже примере:

```
interface IBase
{
        void Base_Func();
}
interface IDerived: IBase
{
        void Derived_Func();
}
class Demo: IDerived
{
        void IBase.BaseFunc()
        {
                // Реализация
        }
```

```
        void IDerived.Derived_Func()
        {
                // Реализация
        }
}
```

Поиск реализации члена интерфейса

До сих пор нами рассматривалась реализация членов интерфейса в классе или структуре. Однако мало реализовать метод или свойство, необходимо еще и обеспечить к нему доступ. Это и будет рассмотрено в данном разделе.

При поиске реализации члена `Member` интерфейса `Interface` в классе используется следующая процедура:

1. В текущем классе производится поиск реализации `Interface.Member`. Если реализация найдена, процедура завершает свою работу.
2. Если реализация не найдена и базовый класс отличается от `object`, он становится текущим и производится переход к действию п. 1.
3. Если поиск явной реализации не привел к успеху, происходит возврат к исходному классу.
4. В текущем классе производится поиск нестатической реализации `Member`. Если реализация найдена, процедура завершает свою работу.
5. Если реализация не найдена и базовый класс отличается от `object`, он становится текущим и производится переход к действию п. 4.
6. Если и этот поиск не привел к успеху, выдается сообщение об ошибке.

Процедура поиска в структуре намного проще, поскольку она не имеет базовой структуры, и поиск производится только в рамках исходной структуры.

Эта операция производится для всех членов всех интерфейсов, перечисленных в списке базовых, независимо от того, используется искомый член в приложении или нет. При этом надо помнить, что реализованы должны быть не только методы, определенные в данном интерфейсе, но и методы всех его базовых интерфейсов.

При поиске реализации считается, что член класса `Realized` реализует член интерфейса `Desirable`, если:

- методы `Desirable` и `Realized` имеют одинаковое имя, тип и список формальных параметров;
- свойства `Desirable` и `Realized` имеют одинаковое имя и тип, и свойство `Realized` имеет те же процедуры доступа, что и свойство `Desirable` (если это неявная реализация, свойство `Realized` может иметь дополнительные процедуры доступа);

- события `Desirable` и `Realized` имеют одинаковое имя и тип;
- индексаторы `Desirable` и `Realized` имеют одинаковый тип и список формальных параметров, а индексатор `Realized` имеет те же процедуры доступа, что и свойство `Desirable` (если это неявная реализация, свойство `Realized` может иметь дополнительные процедуры доступа).

Рассматривая процедуру поиска реализации члена интерфейса необходимо помнить, что:

- явная реализация члена интерфейса, как собственная, так и унаследованная, имеет приоритет перед другими членами класса или структуры;
- при поиске реализации члена интерфейса не рассматриваются члены класса или структуры, имеющие модификатор права доступа `private`, `protected` или `static`.

Рассмотрим пример:

```
interface IFirst
{
        void Some_Func();
}
class Demo: IFirst
{
        void First.Some_Func()
        {
                // Реализация
        }
        void Some_Func();
        {
                // Реализация
        }
}
```

В приведенном примере метод интерфейса будет реализован методом `First.Some_Func`, поскольку явная реализация имеет приоритет.

Если несколько базовых интерфейсов имеют одинаковые члены, то они могут быть реализованы одним членом класса, как это продемонстрировано в примере, приведенном ниже, хотя ничто не мешает использовать явную реализацию для разделения этих членов:

```
interface IFirst
{
        void Some_Func();
}
```

```
interface ISecond
{
        void Some_Func();
}
class Demo: IFirst, ISecond
{
        void Some_Func();
        {
                // Реализация
        }
}
```

При реализации интерфейсов, содержащих скрытые члены, в классе или структуре обязательно следует использовать явную реализацию для этих членов. Исключение может быть сделано только для одного члена, поскольку явная реализация всех остальных конкурирующих членов сделает его неявную реализацию однозначной. Рассмотрим иерархию интерфейсов, скрывающую члены:

```
interface IBase
{
        int Some();
}
interface IDerived: IBase
{
        new int Some
        {
                set;
        }
}
```

Ниже приведены корректные формы их реализации. Поскольку скрыт только один член интерфейса `IDerived`, то и его реализация в классе должна иметь, по крайней мере, одну явную реализацию:

```
class First_Class: IDerived
{
        int IBase.Some() { ... }
        int IDerived.Some { set { ... } }
}
class Second_Class: IDerived
{
```

```
        int IBase.Some() { ... }
        public int Some { set { ... } }
}
class Third_Class: IDerived
{
        public int Some() { ... }
        int IDerived.Some { set { ... } }
}
```

Если класс или структура реализует несколько интерфейсов, имеющих один базовый интерфейс, то члены базового интерфейса должны быть реализованы однократно:

```
interface IBase
{
        void Base_Func();
}
interface IFirst: IBase
{
        void First_Func();
}
interface ISecond: IBase
{
        void Second_Func();
}
class Implementation_Class: IBase, IFirst, ISecond
{
        void IBase.Base_Func() { ... }
        void IFirst.First_Func() { ... }
        void ISecond.Second_Func () { ... }
}
```

Члены базового класса могут использоваться при реализации интерфейса в производном классе. Так в приведенном ниже примере для реализации метода `Some_Func` интерфейса `IBase` используется одноименный метод класса `Base_Class`, являющегося базовым для класса `Derived_Class`, в котором реализуется данный интерфейс:

```
interface IBase
{
        void Some_Func();
        void Other_Func();
}
```

```
class Base_Class
{
        public void Some_Func()
        {
                // Реализация первого метода интерфейса
        }
}
class Derived_Class: Base_Class, IBase
{
        public void Other_Func()
        {
                // Реализация второго метода интерфейса
        }
}
```

Наследование реализации интерфейсов

Класс наследует все реализации интерфейсов своего базового класса. Если в производном классе не создается новая реализация интерфейса, то поиск реализации члена интерфейса начинается с того класса, в котором был определен этот базовый интерфейс. Таким образом, в следующем примере

```
interface IFirst
{
        void Some_Func();
}
class Base_Class: IFirst
{
        public void Some_Func()
        {
                // Реализация
        }
}
class Derived_Class: Base_Class
{
        new public void Some_Func()
        {
                // Реализация
        }
        static void Main()
        {
```

```
            Base_Class    b      = new Base_Class();
            Derived_Class d      = new Derived_Class();
            IFirst        fb     = b;
            IFirst        fd     = d;

            b. Some_Func();      // Вызывает Base_Class.Some_Func
            d. Some_Func();      // Вызывает Derived_Class.Some_Func
            fb. Some_Func();     // Вызывает Base_Class.Some_Func
            fc. Some_Func();     // Вызывает Base_Class.Some_Func
        }
}
```

метод `Derived_Class.Some_Func` скрывает метод `Base_Class.Some_Func`, но оставляет без изменения реализацию метода `IFirst.Some_Func`, как `Base_Class.Some_Func`.

Рассмотрим тот же самый пример, но сделаем функцию базового класса виртуальной и перегрузим ее в производном классе:

```
interface IFirst
{
        void Some_Func();
}
class Base_Class: IFirst
{
        public virtual void Some_Func()
        {
                // Реализация
        }
}
class Derived_Class: Base_Class
{
        public override void Some_Func()
        {
                // Реализация
        }
        static void Main()
        {
                Base_Class    b      = new Base_Class();
                Derived_Class d      = new Derived_Class();;
```

```
            IFirst        fb     = b;
            IFirst        fd     = d;

            b. Some_Func();      // Вызывает Base_Class.Some_Func
            d. Some_Func();      // Вызывает Derived_Class.Some_Func
            fb. Some_Func();     // Вызывает Base_Class.Some_Func
            fc. Some_Func();     // Вызывает Derived_Class.Some_Func
        }
}
```

Непосредственная перегрузка явной реализации члена интерфейса невозможна, поскольку соответствующий член класса не может быть объявлен виртуальным, однако ничто не мешает использовать в этой реализации виртуальный член базового класса и перегрузить его в производном классе.

Создание новой реализации интерфейса

Класс, унаследовавший реализацию интерфейса, может создать для него новую реализацию, включив его в свой список базовых интерфейсов. Поскольку поиск реализации члена интерфейса начинается с текущего класса, то новая реализация этого члена делает недоступной его реализацию в базовом классе. При этом нет необходимости заново реализовывать все члены интерфейса, поскольку не реализованные в производном классе члены будут найдены в базовом классе.

Рассмотрим следующий пример:

```
interface IFirst
{
        void First_Func();
        void Second_Func();
        void Third_Func();
        void Fourth_Func();
}
class Base_Class: IFirst
{
        void IFirst.First_Func()     {}
        public void Second_Func()    {}
        void IFirst.Third_Func()     {}
        public void Fourth_Func()    {}
}
```

```
class Derived_Class: Base_Class, IFirst
{
        void IFirst.First_Func()     {}
        public void Second_Func()    {}
}
```

В приведенном примере методы интерфейса `IFirst` реализуются методами `Derived_Class.IFirst.First_Func`, `Derived_Class.Second_Func`, `Base_Class.IFirst.Third_Func` и `Base_Class.Fourth_Func`.

Новые реализации могут создаваться не только для методов основного интерфейса, но и для методов его базовых интерфейсов.

Абстрактные классы и интерфейсы

Поскольку тип интерфейса является абстрактным, реализация методов интерфейса может производиться и в абстрактных классах. Если класс является абстрактным, то и его методы или, по крайней мере некоторые из них, тоже являются абстрактными. Ниже приведен пример такой реализации:

```
interface IFirst
{
        void Some_Func();
}
abstract class Abstract_Class: IFirst
{
        public abstract void Some_Func();
}
```

Реализация метода `Abstract_Class.Some_Func` должна быть перегружена в производном, неабстрактном классе.

Хотя явные реализации методов интерфейса не могут быть абстрактными, они могут быть помещены в абстрактный класс. При этом для реализации метода интерфейса используется вспомогательный абстрактный метод класса, как это показано в примере, приведенном ниже:

```
interface IFirst
{
        void Some_Func();
}
abstract class Abstract_Class: IFirst
{
        void   IFirst.Some_Func()    { Abstract_Func(); }
        protected abstract void Abstract_Func();
}
```

Как уже говорилось, абстрактные классы и интерфейсы имеют много общего, поэтому следует хорошо разобраться в том, в каких ситуациях следует использовать абстрактные классы, а в каких — интерфейсы.

Одним из основных свойств класса как объекта является *инкапсуляция*: объединение программного кода и данных, используемых для решения одной конкретной задачи. Поэтому классы, в том числе и абстрактные, следует использовать при решении конкретных задач.

В свою очередь, интерфейсы создавались для облегчения взаимодействия различных компонентов, поэтому в них изначально закладывалась универсальность использования. Следовательно, интерфейсы целесообразно использовать в тех случаях, когда необходимо создать некое общее описание для широкого круга слабо связанных между собой объектов.

Использование интерфейсов

При создании сложных приложений довольно трудно гарантировать, что в имеющемся программном коде содержится реализация требуемого интерфейса. Поэтому перед обращением к нему следует проверить, поддерживается ли данный интерфейс соответствующим объектом. Рассмотрим это на примере:

```
using System;

interface IDemo
{
        void   SomeFunc();
}

class CBase
{
        void SomeFunc()
        {
                Console.WriteLine( "Base Class" );
        }
}

class CDemo : CBase, IDemo
{
        public void SomeFunc()
        {
                Console.WriteLine( "Realization" );
        }
}
```

```
class Test
{
        public static void Main()
        {
                CBase[]Array = new CBase[2];

                Array[0] = new CDemo();
                Array[1] = new CBase();

                foreach( CBase demo in Array )
                {
                        if( demo is IDemo)
                        {
                                ((IDemo)demo).SomeFunc();
                        }
                }
        }
}
```

В главном методе этого приложения создается массив, содержащий два объекта, из которых только один реализует интерфейс `IDemo`. После этого организуется цикл, в котором для каждого элемента массива с использованием операции `is` производится проверка, реализован ли в данном объекте интерфейс `IDemo`. Если этот интерфейс реализован, тип объекта преобразуется к типу интерфейса, и для него вызывается метод `SomeFunc`.

В данном примере допустимость преобразования проверяется дважды: операцией `is` и в процессе самого преобразования. Последняя из этих проверок явно избыточна, поскольку успешность выполнения операции преобразования типа гарантирована предыдущей проверкой. Эта проблема может быть решена с использованием операции `as`. Для демонстрации ее использования перепишем предыдущий пример:

```
using System;

interface IDemo
{
        void   SomeFunc();
}

class CBase
{
        void SomeFunc()
        {
```

```
                Console.WriteLine( "Base Class" );
        }
}

class CDemo : CBase, IDemo
{
        public void SomeFunc()
        {
                Console.WriteLine( "Realization" );
        }
}


class Test
{
        public static void Main()
        {
                CBase[]Array = new CBase[2];

                Array[0] = new CDemo();
                Array[1] = new CBase();

                foreach( CBase demo in Array )
                {
                        IDemo  temp   = demo as IDemo;

                        if( temp != null)
                        {
                                temp.SomeFunc();
                        }
                }
        }
}
```

Операция `as` проверяет наличие явного преобразования типа левого операнда к типу его правого операнда и выполняет его в случае успешного завершения проверки. В ином случае результатом этой операции является значение `null`.

При использовании интерфейсов следует помнить о том, что в них отсутствует проверка версий. Это существенно затрудняет внесение в них изменений и требует детальной проработки всех возможностей и областей предполагаемого использования еще в процессе создания интерфейсов.

Глава 12

Атрибуты

При работе с программными единицами любого языка возникает необходимость связать с ними определенную информацию, определяющую их свойства. Например, для задания режима доступа методов класса используются модификаторы режима доступа `public`, `protected`, `internal` и `private`. Однако во многих случаях пользователю необходимо обеспечить возможность связывания с некоторыми программными единицами нестандартной информации и ее получения в процессе выполнения программы. В большинстве языков программирования решение этой задачи полностью возлагается на разработчика. Например, если пользователь захочет связать с некоторой переменной класса некоторое поле базы данных, то ему придется самостоятельно создавать весь необходимый для этого инструментарий. В C++ для этого могут использоваться макросы, однако их использование не регламентировано никакими внешними соглашениями, что чревато возникновением ошибок, не контролируемых средой программирования. Кроме того, макросы тоже должен создавать разработчик.

В среде CLR для решения этой задачи используются *атрибуты*, которые можно рассматривать как примечания к элементам исходного текста программы (типам и их членам). Например, приложение может определить атрибут `AuthorAttribute`, содержащий имя автора данной программной единицы.

Для хранения такой дополнительной информации используются классы атрибутов. Использование атрибутов позволяет управлять процессом выполнения приложения, передавать информацию об объекте или описывать его структуру. Информация атрибутов, относящихся к объекту, хранится совместно с его метаданными.

Классы атрибутов

Классы атрибутов позволяют создавать новые типы атрибутов, которые могут использоваться при объявлении программных единиц. Все классы атрибутов являются прямыми или косвенными потомками абстрактного класса `System.Attribute`. На объявление класса атрибута накладываются следующие ограничения:

- если класс атрибута не является абстрактным, то он должен быть открытым (`public`);

- все типы, в которые вложен неабстрактный класс атрибута, должны быть открытыми;
- неабстрактный класс атрибута должен иметь, по крайней мере, один открытый конструктор;
- все формальные параметры всех открытых конструкторов класса атрибута должны иметь тип параметра атрибута.

По соглашению, имена всех классов атрибутов должны заканчиваться словом `Attribute`. При использовании атрибута это слово может опускаться.

Для определения использования класса атрибута используется атрибут `AttributeUsage`, содержащий позиционные именованные параметры, позволяющие классу атрибута определить, в каких объявлениях он может использоваться. Например, объявление

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class DemoAttribute: System.Attribute
{
        // Реализация класса атрибута
}
```

определяет, что класс атрибута `DemoAttribute` может использоваться в объявлениях классов и интерфейсов. Например:

```
[Demo] interface MyInterface
{
        // Описание интерфейса
}
[Demo] class MyClass1
{
        // Реализация класса
}
```

В приведенном примере имя класса атрибута `DemoAttribute` сокращено до `Demo`., поскольку, как уже говорилось выше, при использовании имени класса атрибута его обязательная составляющая `Attribute` может опускаться.

Атрибут `AttributeUsage` имеет именованный параметр `AllowMultiple`, определяющий возможность многократного использования этого атрибута в определении одной программной единицы. В приведенном ниже примере определен класс атрибута `LanguageAttribute`, определяющий языки, которые поддерживает данный класс. Поскольку класс может работать с несколькими языками, этот атрибут может использоваться многократно:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class LanguageAttribute: System.Attribute
{
```

```
        public LanguageAttribute(string value);
}
```

Ниже продемонстрировано использование данного атрибута при определении класса:

```
[Language("English"), Author("Russian")]
class Translate
{
        // Реализация класса
}
```

Позиционные и именованные параметры

Классы атрибутов могут содержать *позиционные и именованные параметры*. Каждый открытый конструктор класса атрибута определяет корректную последовательность позиционных параметров класса атрибута. Каждое открытое (`public`) нестатическое поле или свойство, доступное как для чтения, так и для записи, определяет именованный параметр класса атрибута. Рассмотрим это на примере:

```
[AttributeUsage(AttributeTargets.Class]
public class MyAttribute: System.Attribute
{
        public MyAttribute(int Count)       // Позиционный параметр Count
        {
                Value = Count;
        }
        public string Counter        // Именованный параметр
        {
                get { ... }
                set { ... }
        }
        int    Value;
}
```

В классе `MyAttribute` определен один позиционный параметр `Count` и один именованный параметр `Counter`. Поле `Value` не является именованным параметром, поскольку является закрытым (`private`).

Типы параметров атрибутов

Набор типов позиционных и именованных параметров класса атрибутов существенно ограничен. Этот набор включает в себя следующие типы:

- `Bool`;
- `Char`;
- `Float`;
- `Long`;
- `String`;
- `System.Type`;
- `Byte`;
- `Double`;
- `Int`;
- `Short`;
- `Object`;
- открытые (`public`) перечислимые типы, вложенные (если они вложены) в открытые типы.

Использование в классах атрибутов других типов для позиционных или именованных параметров является ошибкой. Так, например, в качестве этих параметров не могут использоваться объекты классов.

Определение атрибутов

Атрибут содержит дополнительную информацию об объявляемой программной единице. Атрибуты могут включаться в объявления типов, членов классов, членов интерфейсов, членов перечислимых типов, процедур доступа к свойствам и формальных параметров.

Атрибуты определяются в специальных секциях, каждая из которых заключается в квадратные скобки. В одни квадратные скобки могут быть помещены несколько атрибутов, разделяемых запятой. Порядок расположения секций атрибутов и порядок атрибутов в каждой секции не имеет никакого значения. Таким образом, спецификации `[FirstAttribute][SecondAttribute]`, `[FirstAttribute,SecondAttribute]`, `[SecondAttribute][FirstAttribute]` и `[SecondAttribute,FirstAttribute]` полностью эквивалентны.

Атрибут состоит из своего имени и необязательного списка позиционных и именованных аргументов. Если атрибут содержит неименованные параметры, то ни один из них не может быть пропущен при его использовании (если класс атрибута имеет несколько открытых конструкторов, то набор позиционных атрибутов должен соответствовать набору формальных параметров одного из этих конструкторов). Позиционные аргументы (если они указаны) должны располагаться перед именованными аргументами. Позиционные аргументы представляют собой выражения, в то время как именованные аргументы представляют собой имя аргумента, за которым следуют знак равенства и выражение.

В качестве имени атрибута может использоваться зарезервированное имя атрибута или имя одного из классов атрибутов. Атрибут, не допускающий мно-

гократного использования, должен использоваться в каждом объявлении только один раз. Так приведенный ниже пример содержит ошибку, поскольку

```
[AttributeUsage(AttributeTargets.Class)]
public class DemoAttribute: System.Attribute
{
        int    value;
        public DemoAttribute(int value)
        {
                this.value = value;
        }
        public int Value
        {
                get { ... }
        }
}
[Demo( 1 )]
[Demo( 2 )]    // Ошибка! Атрибут уже определен.
public class Some_Class
{
        // Реализация класса
}
```

К выражениям, используемым в аргументах атрибутов, предъявляются следующие требования:

- оно должно иметь определенный в предыдущем разделе тип параметров атрибутов;
- в процессе компиляции это выражение должно выступать как:
  - константное выражение;
  - объект `System.Type`;
  - одномерный массив выражений, отвечающих данным требованиям.

Следует отметить, что приведенное определение допускает использование и многомерных массивов выражений.

Объекты атрибутов

Объекты атрибутов используются для доступа к атрибутам в процессе выполнения приложения. Объект атрибута представляет собой объект класса атрибута, инициализированный своими позиционными и именованными аргументами. Обращение к объектам атрибутов производится как в процессе компиляции, так и в процессе выполнения приложения.

Компиляция атрибутов

Компиляция атрибутов производится следующим образом:

- С использованием операции `new` создается новый объект класса атрибута, конструктору которого передается список позиционных аргументов, указанных в данном атрибуте. Если не может быть найден соответствующий конструктор класса, или найденный конструктор не является открытым (`public`), компилятор выдает сообщение об ошибке.
- Для каждого именованного аргумента, указанного в данном атрибуте, в описании соответствующего класса атрибута производится поиск одноименного открытого поля или свойства, доступного как для чтения, так и для записи. Если такое поле или свойство не будет обнаружено, компилятор выдает сообщение об ошибке.
- Для последующего вызова атрибута в процессе выполнения приложения сохраняются: класс атрибута, найденный конструктор класса, список позиционных аргументов и список именованных аргументов.

Вызов объектов атрибутов

Вызов объекта атрибута в процессе выполнения программы производится следующим образом:

- С использованием выбранного компилятором конструктора класса атрибута создается объект данного класса. В процессе работы конструктора может быть вызвано исключение, свидетельствующее о возникновении ошибки.
- После создания объекта класса для всех именованных аргументов выполняются следующие действия:
  - если идентификатору именованного аргумента в открытом объекте не соответствует нестатическое открытое поле или свойство, используемое как для чтения, так и для записи, вызывается исключение;
  - если этот идентификатор соответствует полю, то этому полю присваивается значение выражения, указанного в данном аргументе;
  - если этот идентификатор соответствует свойству, то для сохранения в данном свойстве значения выражения, указанного в данном аргументе, вызывается соответствующая процедура доступа.
- В результате выполнения перечисленных выше операций создается объект атрибута, инициализированный своими позиционными и именованными аргументами.

Создание пользовательских атрибутов

Как уже говорилось выше, чтобы создать пользовательский атрибут, необходимо определить класс атрибута, являющийся прямым или косвенным потомком абстрактного класса `System.Attribute`. В качестве примера рассмотрим атрибут, хранящий номер лицензии и номер версии класса или структуры:

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct)]
public class License : System.Attribute
{
        public License (int number)
        {
                this.number    = number;
                version        = 1.0;
        }
        int    number;
        public double version;
        public int    Get()  { return number;}
}
```

Данный класс атрибута содержит закрытое поле `number`, содержащее номер лицензии, и открытое поле `version`, содержащее номер версии. Поле `number` заполняется в конструкторе класса атрибута значением его позиционного аргумента, а поле `number` может быть явным образом изменено в процессе работы с атрибутом.

Для предотвращения коллизий имен все атрибуты имеют имена, заканчивающиеся словом `Attribute`. Если это не так, то это окончание подставляется автоматически. Поэтому данный атрибут может быть вызван как по имени `License`, так и по имени `LicenseAttribute`. Аналогично использованный в примере атрибут `AttributeUsage` имеет полное имя `System.AttributeUsageAttribute`.

Получение информации об атрибуте

В предыдущем разделе был создан атрибут для хранения номера лицензии. Однако возможности записи такого ключа недостаточно для полноценной работы с ним: информацию необходимо не только записывать, но и считывать. Для получения хранящейся в атрибутах информации в C# использует-

ся система отражения. Ключевым компонентом этой системы является метод GetCustomAttributes, возвращающий массив объектов, представляющих собой эквиваленты исходного кода атрибутов, создаваемые в процессе выполнения программы.

При этом атрибут

```
[ (12345, version=2.5)] class Some_Class { ... }
```

представляется в виде

```
License anonymousLicenseObject = new License (12345);
anonymousLicenseObject.version = 2.5;
```

Однако этот программный код не исполняется до тех пор, пока класс Some_Class не запросит данный атрибут. При вызове метода GetCustomAttributes в классе Some_Class создается и инициализируется объект License, как это показано выше. Если класс имеет несколько атрибутов, все их объекты создаются одновременно. Все созданные объекты помещаются в массив, который метод GetCustomAttributes и возвращает вызвавшему его классу. Для получения информации, хранящейся в атрибуте, необходимо найти среди всех элементов массива элемент, имеющий соответствующий тип, и извлечь хранящуюся в нем информацию. Ниже приведен пример использования определенного выше атрибута License:

```
[License(12345, version=2.5)]
class Some_Class
{
        public void WriteLicense()
        {
                object[] attrs = typeof(Some_Class).GetCustomAttributes();
                for (int i=0; i<attrs.Length; ++i)
                {
                        if (attrs[i] is License)
                        {
                                License l = (License)attrs[i];
                                Console.WriteLine( "Version = {0} License No
                                                {1}", l.version, l.Get() );
                        }
                }
        }
}
```

Открытый метод WriteLicense класса Some_Class позволяет вывести информацию о версии данного класса и о номере его лицензии.

Глобальные атрибуты

Большинство атрибутов привязано к конкретным элементам алгоритмического языка, например, к классам или к их методам, но некоторые из них являются глобальными и относятся ко всему ансамблю или модулю. Если глобальные атрибуты относятся к ансамблю, в квадратных скобках перед их списком помещается ключевое слово `assembly`, за которым следует двоеточие. Если же они относятся к модулю, то перед их списком помещается ключевое слово `module`, за которым также следует двоеточие. Глобальные атрибуты размещаются в тексте программы после любой директивы `using` верхнего уровня и перед объявлением любого типа или пространства имен. Они могут размещаться в различных исходных файлах проекта.

В NGWS определены следующие атрибуты ансамбля:

- `System.Reflection.AssemblyConfigurationAttribute`;
- `System.Runtime.CompilerServices.AssemblyCultureAttribute`;
- `System.Reflection.AssemblyDescriptionAttribute`;
- `System.Runtime.CompilerServices.AssemblyKeyFileAttribute`;
- `System.Runtime.CompilerServices.AssemblyKeyNameAttribute`;
- `System.Runtime.CompilerServices.AssemblyDelaySignAttribute`;
- `System.Reflection.AssemblyTitleAttribute`;
- `System.Runtime.CompilerServices.AssemblyVersionAttribute`.

Зарезервированные атрибуты

Некоторые имена атрибутов в C# зарезервированы для внутренних нужд. С одним из этих атрибутов — атрибутом `AttributeUsage` — мы уже познакомились в предыдущих разделах. Эти атрибуты используются для:

- задания свойств пользовательских атрибутов;
- определения режима трансляции метода;
- взаимодействия с другими приложениями;
- сохранения классов и структур в потоках;
- обозначения устаревших компонент.

Ниже будут описаны атрибуты, принадлежащие каждой из этих групп.

Задания свойств пользовательских атрибутов

Для задания свойств пользовательских атрибутов используется уже упоминавшийся выше атрибут `AttributeUsage`.

AttributeUsage

```
[AttributeUsage( validon <>, AllowMultiple=allowmultiple <>, Inher-
ited=inherited <> )]
```

Аргументы

- `Validon` — определяет элементы языка, к которым может быть применен данный атрибут. Представляет собой комбинацию значений перечислимого типа `AttributeTargets`. По умолчанию используется значение `AttributeTargets.All`.
- `Allowmultiple` (необязательный) — имеет тип `bool`. Если он имеет значение `true`, аргумент может многократно использоваться. По умолчанию имеет значение `false` (однократное использование).
- `Inherited` (необязательный) — имеет тип `bool`. Если аргумент `AllowMultiple` имеет значение `true`, в этом аргументе накапливаются унаследованные члены. Если аргумент `AllowMultiple` имеет значение `false`, аргумент перегружается (или заменяется) при наследовании. По умолчанию имеет значение `false` (не наследуется).

Примечание

Атрибут `AttributeUsage` используется при определении классов атрибутов и определяет режимы их использования. Класс, для которого может быть указан атрибут `AttributeUsage`, должен быть прямым или косвенным потомком класса `System.Attribute`. Если это условие не выполняется, компилятор выдает сообщение об ошибке. Полное имя данного атрибута `System.AttributeUsageAttribute`.

Атрибут `AttributeUsage` используется однократно.

Класс `AttributeUsage` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class)]
public class AttributeUsageAttribute: System.Attribute
{
        public AttributeUsageAttribute(AttributeTargets validOn)
        {
                ...
        }
```

```
        public AttributeUsageAttribute(AttributeTargets validOn,
                                       bool allowMultiple, bool inherited)
        {
                ...
        }
        public bool AllowMultiple
        {
                virtual get
                {
                        ...
                }
                virtual set
                {
                        ...
                }
        }
        public bool Inherited
        {
                virtual get
                {
                        ...
                }
                virtual set
                {
                        ...
                }
        }
        public AttributeTargets ValidOn
        {
                virtual get
                {
                        ...
                }
        }
}
```

Атрибуты условной трансляции

Атрибуты условной трансляции используются для обозначения методов, выполнение которых зависит от указанных директив препроцессора. К этой группе относится только атрибут `conditional`.

conditional

```
[conditional(conditionalSymbol <>)]
```

Аргументы

`ConditionalSymbol` — имеет тип `string` и содержит идентификатор препроцессора.

Примечание

Атрибут `conditional` отмечает условный метод, то есть метод, выполнение которого зависит от определенного идентификатора препроцессора. Данный атрибут содержит условие, имеющее форму идентификатора препроцессора. Условный метод вызывается только в том случае, если соответствующий символ определен. В противном случае вызов метода опускается. Полное имя данного атрибута `System.Diagnostics.ConditionalAttribute`.

Условные методы должны отвечать следующим требованиям:

- являться членами класса. Если атрибут `conditional` используется для метода интерфейса, компилятор выдает сообщение об ошибке;
- не возвращать значения (иметь тип возвращаемого значения `void`);
- не иметь модификатора `override`. Условный метод может иметь модификатор `virtual`. Все перегрузки данного метода также являются условными методами, поэтому нет необходимости указывать для них атрибут `conditional`;
- не может являться реализацией метода интерфейса. В противном случае компилятор выдаст сообщение об ошибке;
- не может использоваться для создания делегата.

Класс атрибута `conditional` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
public class ConditionalAttribute: System.Attribute
{
        public ConditionalAttribute(string conditionalSymbol)
        {
                ...
        }
        public string ConditionalSymbol
        {
                get
                {
                        ...
                }
        }
}
```

Атрибут conditional используется многократно.

При вызове условного метода наличие или отсутствие в точке вызова символа препроцессора, определенного в аргументе conditionalSymbol, определяет, будет ли вызов этого метода включен в окончательный текст программы. Если этот символ определен, вызов метода включается в программу, в противном случае вызов этого метода опускается. Использование условных методов представляет собой более изящную альтернативу помещению вызова метода между директивами препроцессора #if conditionalSymbol и #endif.

Если метод имеет несколько атрибутов conditional, вызов метода включается в программу, если определен хотя бы один из символов условия (другими словами, условия объединятся операцией логического ИЛИ):

```
[conditional("First"), conditional("Second")]
public static void Some_Func()
{
        ...
}
```

Для объединения условий операцией логического И используется вложение методов, как это показано ниже:

```
[conditional("First")]
public static void Some_Func()
{
        Temp_Func();
}
[conditional("Second")]
static void Temp_Func()
{
        // Тело метода
}
```

Поскольку метод Temp_Func является закрытым (private), то единственным способом его вызова является обращение к методу Some_Func. Таким образом, тело метода будет выполнено только в том случае, если одновременно будут выполнены условия First и Second.

Ниже приведен пример использования атрибута conditional.

```
#define DEBUG
class First_Class
{
        [conditional("DEBUG")]
```

```
        public static void Some_Func()
        {
                Console.WriteLine("Debug version");
        }
}
class Second_Class
{
        public static void Test()
        {
                First_Class.Some_Func();
        }
}
```

Поскольку в данной программе определен идентификатор `DEBUG`, метод `Some_Func` класса `First_Class` выполняется при его вызове в методе `Test` класса `Second_Class` и выводит сообщение о том, что данная версия является отладочной. В этом примере идентификатор `DEBUG` определен до описания обоих классов. При работе с атрибутом `conditional` следует помнить о том, что условие вызова метода проверяется в точке вызова данного метода. Рассмотрим это на примере:

```
// Файл First_Class.cs
class First_Class
{
        [conditional("DEBUG")]
        public static void Some_Func()
        {
                Console.WriteLine("Debug version");
        }
}
// Конец файла First_Class.cs

// Файл Second_Class.cs
#define DEBUG
class Second_Class
{
        public static void Test
        {
                First_Class.Some_Func();
        }
}
```

```
// Конец файла Second_Class.cs

// Файл Third_Class.cs
#undef DEBUG
class Third_Class
{
        public static void Test
        {
                First_Class.Some_Func();
        }
}
// Конец файла Third_Class.cs
```

Все классы в рассмотренном примере расположены в отдельных файлах, поскольку синтаксис C# не допускает использования директив определения идентификаторов после определения первой именованной области или типа в файле. В данном примере определенность идентификатора `DEBUG` в первом файле не имеет значения, поскольку в нем условный метод определяется, но не используется. Во втором файле идентификатор `DEBUG` определен. Поэтому при вызове метода `Test` класса `Second_Class` будет выведено сообщение о том, что программа работает в отладочном режиме. Поскольку в третьем файле идентификатор `DEBUG` не определен, при вызове метода `Test` класса `Third_Class` не будет произведено никаких действий.

Поскольку все перегрузки условного метода также являются условными методами, использующими то же самое условие, при их вызове не возникает конфликтов. В приведенном ниже примере в зависимости от определенности идентификатора `DEBUG` на экран выводятся две строки или не выводится ни одной:

```
class Base_Class
{
        [conditional("DEBUG")]
        public virtual void Some_Func()
        {
                Console.WriteLine("Debug version");
        }
}
class Derived_Class : Base_Class
{
        public override void Some_Func()
        {
```

```
            Console.WriteLine("Derived Class");
            base.Some_Func();
        }
        static void Main()
        {
            Derived_Class d      = new Derived_Class();
            d.Some_Func();
        }
}
```

Взаимодействие с другими приложениями

Для взаимодействия с другими приложениями в C# используется широкий набор атрибутов. Эти атрибуты позволяют хранить в классах информацию, необходимую для их идентификации серверами и приемниками. Одной из основных областей применения атрибутов является обеспечение взаимодействия программ, написанных на различных языках, удовлетворяющих требованиям соглашения NGWS, с программами, использующими соглашения COM.

comimport

```
[comimport]
```

Примечание

Атрибут `comimport` используется для определения класса как внешней структуры `coclass` COM. Указание данного атрибута позволяет использовать имена C# для обращения к классам COM. Атрибут `comimport` используется однократно. Полное имя данного атрибута `System.InterOp.ComImportAttribute`.

Класс атрибута `conditional` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class)]
public class COMImportAttribute: System.Attribute
{
        public COMImportAttribute()
        {
            ...
        }
}
```

Класс, имеющий атрибут comimport, должен удовлетворять следующим условиям:

- иметь также атрибут guid, определяющий CLSID импортируемого класса COM. (при отсутствии данного атрибута компилятор выдаст сообщение об ошибке);
- не должен содержать никаких членов(по умолчанию в него будет помещен конструктор класса, не имеющий формальных параметров);
- должен быть прямым потомком класса object.

Создание класса comimport с использованием оператора new эквивалентно вызову функции CoCreateInstance.

Рассмотрим пример использования атрибута comimport:

```
[comimport, guid("867ED504-417A-FCD6-9A78-0782863F93C5")]
class Demo
{
        // Данный класс не может содержать никаких членов
        // Компилятор C# подставит сюда используемый по умолчанию
        // конструктор, не имеющий аргументов
}
class UseDemo
{
        public static void Main()
        {
                // Приведенное ниже выражение эквивалентно вызову функции
                // CoCreateInstance(867ED504-417A-FCD6-9A78-0782863F93C5,
                //                                 NULL, CLSCTX_ALL,
                //                                 IID_IUnknown, &f),
                // возвращающей нулевое значение при возникновении ошибки.
                Demo d = new Demo();
        }
}
```

Создание объекта данного класса приводит к одновременному созданию соответствующего объекта класса COM.

comsourceinterfaces

```
[comsourceinterfaces(val <>)]
```

Аргументы

Val — имеет тип string и содержит список базовых интерфейсов импортируемого coclass.

Примечание

Атрибут `comsourceinterfaces` используется в объявлениях классов для хранения списка базовых интерфейсов импортируемого `coclass`. Данный атрибут используется однократно. Полное имя данного атрибута `System.InterOp.ComSourceInterfacesAttribute`.

Класс атрибута `comsourceinterfaces` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class)]
public class ComSourceInterfacesAttribute: System.Attribute
{
        public ComSourceInterfacesAttribute(string value)
        {
                ...
        }
        public string Value
        {
                get
                {
                        ...
                }
        }
}
```

comvisibility

```
[comvisibility(val <>)]
```

Аргументы

`Val` — имеет тип `ComVisibility` и определяет видимость класса или интерфейса в COM.

Примечание

Атрибут `comvisibility` используется в объявлениях классов и интерфейсов для задания их видимости в COM. Данный атрибут используется однократно. Полное имя данного атрибута `System.InterOp.COMVisibilityAttribute`.

Класс атрибута `comvisibility` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class COMVisibilityAttribute: System.Attribute
{
        public COMVisibilityAttribute(System.Interop.ComVisibility value)
        {
```

```
                ...
        }
        public ComVisibilityAttribute Value
        {
                get
                {
                        ...
                }
        }
}
```

dispid

```
[dispid(val <>)]
```

Аргументы

`Val` — имеет тип `int` и содержит идентификатор диспетчеризации.

Примечание

Атрибут `dispid` определяет идентификатор диспетчеризации COM для метода, поля или свойства. Данный атрибут используется однократно. Полное имя данного атрибута `System.InterOp.DispIdAttribute`.

Класс атрибута `comvisibility` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Field | At-
tributeTargets.Property)]
public class DispIdAttribute: System.Attribute
{
        public DispIdAttribute(int value)
        {
                ...
        }
        public int Value
        {
                get
                {
                        ...
                }
        }
}
```

Этот атрибут обеспечивает интероперабельность COM и обычно определяется для членов интерфейсов. Ниже приведен пример использования атрибута `dispid` при объявлении метода интерфейса:

```
interface ITranslate
{
        [dispid(1010)]
        bool Translate(string word, out string[] translatedString);
}
```

dllimport

```
[dllimport( dllName <>, EntryPoint=entrypoint <>, CharSet=charset <>,
SetLastError=setlasterror <>, ExactSpelling=exactspelling <>, CallingCon-
vention=callingconvention <> )]
```

Аргументы

- `dllName` — имеет тип `string` и определяет имя библиотеки динамической компоновки (DLL), в которой содержится реализация импортируемого метода;
- `entrypoint` (необязательный) — имеет тип `string` и определяет имя входной точки в библиотеке динамической компоновки. Если параметр `EntryPoint` не указан, то в качестве входной точки используется имя метода;
- `charset` (необязательный) — имеет тип `CharSet` и определяет набор символов, используемый во входной точке. Если параметр `CharSet` не указан, по умолчанию используется значение `CharSet.Auto`;
- `setlasterror` (необязательный) — имеет тип `bool`. Если данный аргумент имеет значение `true`, в методе сохраняется значение последней ошибки Win32. По умолчанию имеет значение `false` (значение последней ошибки не сохраняется, но быстродействие при этом повышается);
- `exactspelling` (необязательный) — имеет тип `bool`. Если данный аргумент имеет значение `true`, содержимое аргумента `EntryPoint` должно полностью соответствовать имени указанной точки входа. По умолчанию имеет значение `false`, позволяющее использовать при поиске имени точки входа некоторые эвристические методы;
- `callingconvention` (необязательный) — имеет тип `CallingConvention`. Определяет соглашение о вызове функций, используемое для входной точки библиотеки динамической компоновки. По умолчанию имеет значение `CallingConvention.Winapi`.

Примечание

Атрибут `dllimport` используется для указания расположения библиотеки динамической компоновки, содержащей реализацию внешнего метода. Этот атрибут используется однократно. Полное имя данного атрибута `System.InterOp.DllImportAttribute`.

Класс атрибута `dllimport` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Method)]
public class DllImportAttribute: System.Attribute
{
        public DllImportAttribute(string dllName)
        {
                ...
        }
        public CallingConvention CallingConvention;
        public CharSet CharSet;
        public string DllName
        {
                get
                {
                        ...
                }
        }
        public string EntryPoint;
        public bool ExactSpelling;
        public bool SetLastError;
}
```

Если метод имеет атрибут `dllimport`, то он должен, также иметь модификатор `extern`, поскольку данный атрибут используется для определения библиотеки динамической компоновки (DLL), содержащей реализацию внешнего (`extern`) метода.

В приведенном ниже примере использование атрибута `dllimport` позволяет автоматически определить, какую именно из функций `MessageBoxA` или `MessageBoxW` следует использовать при обращении к функции `MessageBox`. Это достигается тем, что аргумент `ExactSpelling` имеет значение, установленное по умолчанию (`false`), позволяющее выбирать реализацию функции в зависимости от используемой платформы. При маршалинге строк будет автоматически выбираться формат ANSI или Unicode (поскольку аргумент `CharSet` по умолчанию имеет значение `CharSet.Auto`).

```
class DllimportDemo
{
        [dllimport("user32.dll")]
        public static extern int MessageBox(int h, string m, string c,
int type);
        public static int Main()
        {
                return MessageBox(0, "Helo, I am C#!", "Demo", 0);
        }
}
```

globalobject

```
[globalobject]
```

Примечание

Атрибут `globalobject` определяет класс как глобальный (appobject) класс COM. Данный атрибут используется однократно. Полное имя данного атрибута `System.InterOp.GlobalObjectAttribute`.

Класс атрибута `globalobject` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class)]
public class GlobalObjectAttribute: System.Attribute
{
        public GlobalObjectAttribute()
        {
                ...
        }
}
```

guid

```
[guid <>(guid <>)]
```

Аргументы

`Guid` — имеет тип `string` и определяет глобально-уникальный идентификатор (GUID).

Примечание

Атрибут `guid` используется при определении типов классов, интерфейсов, перечисляемых типов и делегатов. Данный атрибут определяет для указанных типов глобально-уникальный идентификатор (GUID). Эта информация используется прежде всего для интероперабельности COM. Полное имя данного атрибута `System.InterOp.GuidAttribute`.

Класс атрибута `guid` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface |
AttributeTargets.Enum | AttributeTargets.Delegate | AttributeTar-
gets.Struct)]
public class GuidAttribute: System.Attribute
{
        public GuidAttribute(string uuid)
        {
                ...
        }
        public Guid Value
        {
                get
                {
                        ...
                }
        }
}
```

Атрибут `guid` используется однократно и имеет один аргумент. GUID представляет собой строку формата `XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX`, где `X` представляет собой шестнадцатиразрядное число. Если строка, указанная в аргументе данного атрибута имеет формат, отличающийся от указанного выше, компилятор выдает сообщение об ошибке.

Ниже приведен пример использования данного атрибута при определении типа интерфейса:

```
[guid("942367D6-9C86-734C-1834-A95BF504E9F1")]
interface ITranslate
{
        bool Translate(string word, out string[] translatedString);
}
```

hasdefaultinterface

```
[hasdefaultinterface]
```

Примечание

Атрибут `hasdefaultinterface` используется в описаниях классов для указания того, что определяемый класс имеет интерфейс, используемый по умолчанию. Данный атрибут используется однократно. Полное имя данного атрибута `System.InterOp.HasDefaultInterfaceAttribute`.

Класс атрибута `hasdefaultinterface` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class)]
public class HasDefaultInterfaceAttribute: System.Attribute
{
        public HasDefaultInterfaceAttribute()
        {
        ...
        }
}
```

importedfromcom

```
[importedfromcom(val <>)]
```

Аргументы

`Val` — имеет тип `string` и содержит имя библиотеки типов COM из которой был импортирован данный модуль.

Примечание

Атрибут `importedfromcom` используется при определении модулей и указывает на то, что модуль был импортирован из библиотеки типов COM. Данный атрибут используется однократно. Его полное имя `System.InterOp.ImportedFromCOMAttribute`.

Класс атрибута `importedfromcom` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Module)]
public class ImportedFromCOMAttribute: System.Attribute
{
        public ImportedFromCOMAttribute(string value)
        {
        ...
        }

        public string Value
        {
                get
                {
                        ...
                }
        }
}
```

in

```
[in]
```

Примечание

Атрибут `in` используется для указания формальных параметров, передаваемых вызывающим приложением вызываемому методу. Эта информация используется при пользовательском маршалинге. Этот атрибут используется однократно. Его полное имя `System.InterOp.InAttribute`.

Класс атрибута `in` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Parameter)]
public class InAttribute: System.Attribute
{
        public InAttribute()
        {
                ...
        }
}
```

Атрибут `in` может использоваться совместно с атрибутом `out`. Если для аргумента не указан ни атрибут `in`, ни атрибут `out`, его маршалинг зависит от модификаторов аргумента: модификатор `ref` подразумевает комбинацию атрибутов `[in, out]`, модификатор `out` — атрибут `[out]`, а отсутствие любого модификатора — атрибут `[in]`.

В качестве примера использования атрибута `[in]` рассмотрим гипотетический интерфейс переводчика, в котором переводимое слово имеет атрибут `[in]`, а массив возможных вариантов перевода — атрибут `[out]`.

```
interface ITranslate
{
        bool Translate([in]string word, [out]string[] translatedString);
}
```

interfacetype

```
[interfacetype(val <>)]
```

Аргументы

`Val` — имеет тип `ComInterfaceType` и определяет тип интерфейса.

Примечание

Атрибут `interfacetype` используется при определении класса интерфейса для задания типа интероперабельности интерфейса COM: интерфейс диспетчеризации, двойной или производный от интерфейса `IUnknown`. Его полное имя `System.InterOp.InterfaceTypeAttribute`.

Класс атрибута `interfacetype` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Interface)]
public class InterfaceTypeAttribute: System.Attribute
{
        public InterfaceTypeAttribute(System.Interop.ComInterfaceType
                                      value)
        {
                ...
        }
        public System.Interop.ComInterfaceType Value
        {
                get
                {
                        ...
                }
        }
}
```

Атрибут `interfacetype` используется однократно. Интерфейс, имеющий атрибут `interfacetype`, должен также иметь атрибут `guid`.

Ниже приведен пример использования атрибута `interfacetype`:

```
[comimport, guid("5D8861B1-1AC1-13C2-B1CD-0110A10BF711"),
                  interfacetype(ComInterfaceType.Dual)]
interface IPlay              // Данный интерфейс не может иметь базовых
                              // интерфейсов
{
        [returnshresult(true)] void Run();
        [returnshresult(true)] void Pause();
        [returnshresult(true)] void Stop();
        ...
}
```

iscomregisterfunction

```
[iscomregisterfunction]
```

Примечание

Атрибут `iscomregisterfunction` используется в описаниях методов для указания того, что для его вызова должен быть использован процесс регистрации COM. Этот атрибут используется однократно. Его полное имя `System.InterOp.IsCOMRegisterFunctionAttribute`.

Класс атрибута `iscomregisterfunction` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Method)]
public class IsCOMRegisterFunctionAttribute: System.Attribute
{
        public IsComRegisterFunctionAttribute()
        {
                ...
        }
}
```

marshal

```
[marshal( unmanagedType, Cookie=cookie <>, IID=iid <>,
Marshaler=marshaler <>, SubType=subtype <>, Size=size <>)]
```

Аргументы

- `unmanagedtype` — определяет неуправляемый формат маршалинга;
- `cookie` (необязательный) — имеет тип `string`. Определяет передаваемый маршалеру файл `cookie`;
- `iid` (необязательный) — имеет тип `string`. Задает GUID для типов `NativeType.Interface`. Если аргумент `unmanagedtype` имеет значение `UnmanagedType.Interface`, то данный аргумент содержит GUID;
- `marshaler` (необязательный) — имеет тип `string`. Определяет класс маршалинга;
- `subtype` (необязательный) — имеет тип `UnmanagedType`. Определяет дополнительный тип для типов `UnmanagedType.Ptr` и `UnmanagedType.FixedArray` (по другим сведениям — для типов `NativeType.Ptr` и `NativeType.FixedArray`);
- `size` (необязательный) — имеет тип `int`. Содержит размер строки или массива фиксированного размера.

Примечание

Атрибут `marshal` используется при описании методов, параметров и свойств для задания формата аргумента, поля или возвращаемого значения метода при неуправляемом (`unmanaged`) маршалинге. Этот атрибут используется однократно. Его полное имя `System.InterOp.MarshalAttribute`.

Класс атрибута marshal имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Parameter |
AttributeTargets.Field)]
public class MarshalAttribute: System.Attribute
{
        public MarshalAttribute(UnmanagedType type)
        {
                ...
        }
        public string Cookie;
        public Guid IID;
        public Type Marshaler;
        public UnmanagedType NativeType
        {
                get
                {
                        ...
                }
        }
        public int Size;
        public UnmanagedType SubType;
}
```

Данный атрибут указывает на способ маршалинга элемента. Например, при маршалинге аргумента типа string могут использоваться типы BStr, LPStr, LPWStr, LPTStr или FixedSysString. Для методов атрибут marshal определяет способ маршалинга его возвращаемого значения.

В приведенном ниже примере метод ChosePlayer принимает один аргумент типа string, в котором передается имя воспроизводимого файла. При маршалинге этот аргумент имеет тип BStr. Другой аргумент, в котором возвращается объект проигрывателя данного файла, имеет тип object, рассматриваемый при маршалинге как экземпляр интерфейса IDispatch. Возвращаемое значение метода PlayerCollection, имеющее тип object, также рассматривается при маршалинге как экземпляр интерфейса Idispatch:

```
[guid("942367D6-9C86-734C-1834-A95BF504E9F1"),
InterfaceType(ComInterfaceType.Dual)]
interface IMediaPlayer // Данный интерфейс не может иметь базовых
                       //интерфейсов
{
```

```
        [returnshresult(true)]
        void ChosePlayer(
                [in, marshal(UnmanagedType.BStr)] string szSourceFile,
                [out, marshal(UnmanagedType.IDispatch)] object ppPlayer
        );
        [returnshresult(true), marshal(UnmanagedType.IDispatch)]
        object PlayerCollection();
}
```

name

```
[name(name <>)]
```

Аргументы

`Name` — имеет тип `string` и содержит имя свойства, используемого индексатором NGWS.

Примечание

Атрибут `name` применяется в объявлениях индексаторов NGWS для указания используемого ими свойства. Если атрибут `name` не указан, свойству назначается идентификатор Item. Если указанный идентификатор не соответствует требованиям, предъявляемым к идентификатором C#, компилятор выдает сообщение об ошибке. Атрибут `name` используется однократно. Полное имя данного атрибута `System.InterOp.NameAttribute`.

Класс атрибута `name` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Indexer)]
public class NameAttribute: System.Attribute
{
        public NameAttribute(string value)
        {
                ...
        }
        public string Value
        {
                get
                {
                        ...
                }
        }
}
```

noidispatch

```
[noidispatch]
```

Примечание

Атрибут `noidispatch` используется в описаниях классов и интерфейсов для указания того, что при их экспорте в COM в качестве базового интерфейса следует использовать интерфейс IUnknown, а не интерфейс IDispatch. Этот атрибут используется однократно. Его полное имя `System.InterOp.NoIDispatchAttribute`.

Класс атрибута `noidispatch` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class NoIDispatchAttribute: System.Attribute
{
        public NoIDispatchAttribute()
        {
                ...
        }
}
```

out

```
[out]
```

Примечание

Атрибут `out` используется для указания аргументов, передаваемых вызываемым методом вызывающему приложению. Эта информация используется при пользовательском маршалинге. Атрибут `out` используется однократно. Его полное имя `System.InterOp.OutAttribute`.

Класс атрибута `out` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Parameter)]
public class OutAttribute: System.Attribute
{
        public OutAttribute()
        {
                ...
        }
}
```

Атрибут `out` может использоваться совместно с атрибутом `in`. Если для аргумента не указан ни атрибут `in` ни атрибут `out`, его маршалинг зависит от модификаторов аргумента: модификатор `ref` подразумевает комбинацию

атрибутов `[in, out]`, модификатор `out` — атрибут `[out]`, а отсутствие любого модификатора — атрибут `[in]`.

В качестве примера использования атрибута `[in]`, рассмотрим гипотетический интерфейс переводчика, в котором переводимое слово имеет атрибут `[in]`, а массив возможных вариантов перевода — атрибут `[out]`:

```
interface ITranslate
{
        bool Translate([in]string word, [out]string[] translatedString);
}
```

predeclared

```
[predeclared out]
```

Примечание

Вызываемым методом вызывающему приложению. Эта информация используется при пользовательском маршалинге. Атрибут `predeclared` используется однократно. Его полное имя `System.InterOp.PredeclaredAttribute`.

Класс атрибута `predeclared` имеет следующее описание:

```
[AttributeUsage(Attribute(AttributeTargets.Class)]
public class PredeclaredAttribute: System.Attribute
{
        public PredeclaredAttribute()
        {
                ...
        }
}
```

returnshresult

```
[returnshresult(value)]
```

Аргументы

`Value` — имеет тип `bool`. Если данный аргумент имеет значение `true`, то объявляемый метод или свойство используют конвенцию вызова COM. В противном случае они используют конвенцию вызова NGWS.

Примечание

Атрибут `returnshresult` используется при объявлении методов и свойств для указания используемой конвенции вызова (COM или NGWS). Этот атрибут используется однократно. Его полное имя `System.InterOp.ReturnsHResultAttribute`.

Класс атрибута `returnshresult` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Property)]
public class ReturnsHResultAttribute: System.Attribute
{
        public ReturnsHResultAttribute(bool value)
        {
                ...
        }
        public bool Value
        {
                get
                {
                        ...
                }
        }
}
```

Метод, имеющий атрибут `returnshresult`, не должен иметь тела. Поэтому данный атрибут может использоваться только в описаниях методов интерфейсов или внешних методов классов, имеющих модификатор `extern`. Если атрибут `returnshresult` используется в описании других методов, компилятор выдает сообщение об ошибке.

Как правило, неуправляемые (unmanaged) интерфейсы COM возвращают значение типа `HRESULT`, содержащее информацию об успешности завершения их работы, а сами результаты работы возвращаются ими через свои аргументы. В отличие от этого методы NGWS возвращают результаты своей работы, а для обработки возникающих ошибок используют исключения. Атрибут `returnshresult` позволяет преобразовать конвенцию вызова COM в конвенцию вызова NGWS. В качестве примера рассмотрим следующий метод COM:

```
interface IDemo
{
        HRESULT Logarithm([in]double x, [out,retval]double *result);
}
```

Возвращаемое значение метода в определении интерфейса MIDL имеет атрибуты `out` и `retval`. Эти атрибуты содержат второй аргумент метода `Logarithm` определенного выше интерфейса `IDemo`. В C# данный интерфейс будет иметь следующий вид:

```
interface IDemo
{
```

```
        HRESULT Logarithm(double x, out double returnValue);
}
```

Вызов данного метода будет иметь вид:

```
double y;
// Предположим, что интерфейс IDemo реализован в объекте exampleObj
HRESULT hr = exampleObj.Logarithm(100.0, out y);
if (hr != 0)
        throw new COMInteropException(hr);
// Продолжение работы приложения
```

Однако существует и более изящное решение, использующее атрибут `returnshresult`. Если в C# объявить данный интерфейс следующим образом:

```
interface IDemo
{
        [returnshresult(true)] double Logarithm(double x);
}
```

то его вызов существенно упрощается и выглядит так:

```
// Предположим, что интерфейс IDemo реализован в объекте exampleObj
double y = exampleObj.Logarithm(100.0);
```

Использование методов, возвращающих результаты своей работы, существенно повышает читабельность программ. То, что в действительности они представляют собой методы COM, возвращающие значения типа `HRESULT`, скрывается от пользователя. При этом в эти методы автоматически вставляются проверка значения `HRESULT` и вызов соответствующего исключения в случае необходимости.

Нестатические методы имеют неявный атрибут `returnshresult(true)`, а статические методы — неявный атрибут `returnshresult(false)`. При работе с неуправляемыми (unmanaged) приложениями нестатические методы соответствуют методам интерфейса COM. Поскольку большинство методов COM возвращают значение `HRESULT`, для них устанавливается атрибут `returnshresult(true)`. Статические методы соответствуют внешним методам `dllimport`, не возвращающим значение `HRESULT`. Поэтому в большинстве случаев могут использоваться значения, установленные по умолчанию. В тех случаях, когда это не так, необходимо в явном виде указать атрибут `returnshresult`. Если дело не касается интероперабельности, аргумент `returnshresult` не оказывает никакого воздействия на метод или свойство, для которого он определен.

structlayout

```
[structlayout(kind, Pack=pack, CharSet=charset, CheckFastMar-
shal=checkfastmarshal)]
```

Аргументы

- `kind` — имеет перечислимый тип `LayoutKind`, определяющий тип расположения: последовательный, объединение или явный;
- `pack` (необязательный) — имеет тип `int` и определяет размер упаковки в байтах. Может иметь значение 1, 2, 4, 8 или 16 (по другим сведениям — любая целая степень числа 2). По умолчанию используется значение 8 (по другим сведениям — 4);
- `charset` (необязательный) — имеет тип `CharSet`, определяющий используемый набор символов и строковых типов. По умолчанию имеет значение `CharSet.Auto`;
- `checkfastmarshal` (необязательный) — имеет тип `bool`. Если данный аргумент имеет значение `true`, то компилятор выдает сообщение об ошибке при использовании некорректного типа. По умолчанию имеет значение `false` (сообщение не выдается).

Примечание

Атрибут `structlayout` используется для задания расположения полей классов и структур для обеспечения интероперабельности с неуправляемыми (`unmanaged`) приложениями. Поля, для которых устанавливается данный атрибут, должны иметь тип структуры. Атрибут `structlayout` используется однократно. Его полное имя `System.InterOp.StructLayoutAttribute`.

Класс атрибута `structlayout` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class StructLayoutAttribute: System.Attribute
{
        public StructLayoutAttribute(LayoutKind kind)
        {
                ...
        }
        public CharSet CharSet;
        public int Pack;
        public LayoutKind StructLayoutKind
        {
                get
                {
                        ...
                }
        }
}
```

Если аргумент `kind` данного атрибута имеет значение `Layout.Sequential`, то для каждого последующего члена назначается такое смещение относительно предыдущего члена, которое обеспечивает отсутствие их перекрытия. При необходимости возникающие при этом пробелы заполняются нулями.

Если аргумент `kind` данного атрибута имеет значение `Layout.Union`, то всем членам назначается нулевое смещение. Эти члены не могут иметь тип ссылки.

Если аргумент `kind` данного атрибута имеет значение `Layout.Explicit`, каждое поле должно иметь атрибут `structoffset`. Указание данного значения аргумента является единственной возможностью использования `structoffset`.

Ниже приведено описание структуры `PDH_RAW_COUNTER`, полностью соответствующей одноименной структуре SDK:

```
[structlayout(LayoutKind.Sequential, pack=8)]
public struct PDH_RAW_COUNTER
{
        public int CStatus;
        public FILETIME TimeStamp;
        public long FirstValue;
        public long SecondValue;
        public int MultiCount;
}
[structlayout(LayoutKind.Sequential, pack=4)]
public struct FILETIME
{
        public int dwLow;
        public int dwHigh;
}
```

structoffset

```
[structoffset(offset)]
```

Аргументы

`Offset` — имеет тип `int` и содержит смещение поля в байтах.

Примечание

Атрибут `structoffset` задает смещения отдельных полей классов и структур для обеспечения интероперабельности с неуправляемыми (unmanaged) приложениями. Этот атрибут используется однократно. Его полное имя `System.InterOp.StructOffsetAttribute`.

Класс атрибута `structlayout` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Field)]
public class StructOffsetAttribute: System.Attribute
{
        public StructOffsetAttribute(int offset)
        {
                ...
        }
}
```

Этот атрибут может использоваться только в том случае, если для класса или структуры, для полей которых он будет использован, указан атрибут `structlayout(LayoutKind.Explicit)`. При использовании данного атрибута не производится никаких проверок, поэтому программист может задавать любые значения смещения.

Ниже приведено описание структуры `PDH_RAW_COUNTER`, полностью соответствующей одноименной структуре SDK:

```
[structlayout(LayoutKind.Explicit)]
public struct PDH_RAW_COUNTER {
        [structoffset(0)] public int CStatus;
        [structoffset(16)] public long FirstValue;
        [structoffset(4)] public FILETIME TimeStamp;
        [structoffset(24)] public long SecondValue;
        [structoffset(32)] public int MultiCount;
}
[structlayout(LayoutKind.Explicit)]
public struct FILETIME {
        [structoffset(0)] public int dwLow;
        [structoffset(4)] public int dwHigh;
}
```

typelibfunc

```
[typelibfunc(val)]
```

Аргументы

`Val` — имеет тип `short` и содержит флаги `typelib`.

Примечание

Атрибут `typelibfunc` используется при определении методов, задавая для них флаги `typelib`, используемые для интероперабельности COM. Этот атрибут используется однократно. Его полное имя `System.InterOp.TypeLibFuncAttribute`.

Класс атрибута `typelibfunc` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Method)]
public class TypeLibFuncAttribute: System.Attribute
{
        public TypeLibFuncAttribute(short value)
        {
                ...
        }
        public short Value
        {
                get
                {
                        ...
                }
        }
}
```

typelibtype

```
[typelibtype(val)]
```

Аргументы

`Val` — имеет тип `short` и содержит флаги `typelib`.

Примечание

Атрибут `typelibtype` используется при определении классов и интерфейсов, задавая для них флаги typelib, используемые для интероперабельности COM. Этот атрибут используется однократно. Его полное имя `System.InterOp.TypeLibTypeAttribute`.

Класс атрибута `typelibtype` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class TypeLibTypeAttribute: System.Attribute
{
        public TypeLibTypeAttribute(short value)
        {
                ...
        }
        public short Value
        {
```

```
                get
                {
                        ...
                }
        }
}
```

typelibvar

```
[typelibvar(val)]
```

Аргументы

`Val` — имеет тип `short` и содержит флаги `typelib`.

Примечание

Атрибут `typelibvar` используется при определении полей, задавая для них флаги `typelib`, используемые для интероперабельности COM. Этот атрибут используется однократно. Его полное имя `System.InterOp.TypeLibVarAttribute`.

Класс атрибута `typelibvar` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Field)]
public class TypeLibVarAttribute: System.Attribute
{
        public TypeLibVarAttribute(short value)
        {
                ...
        }
        public short Value
        {
                get
                {
                        ...
                }
        }
}
```

Сохранение классов и структур в потоках

Атрибуты, используемые при сохранении классов и структур в потоках, позволяют разрешить или запретить эту операцию для конкретного класса или структуры.

nonserialized

```
[nonserialized]
```

Примечание

Атрибут `nonserialized` используется для запрещения сохранения полей и свойств класса или структуры в потоке. Этот атрибут используется однократно. Его полное имя `System.Serialization.NonSerializedAttribute`.

Класс атрибута `nonserialized` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property)]
public class NonSerializedAttribute: System.Attribute
{
        public NonSerializedAttribute()
        {
                ...
        }
}
```

Если класс имеет атрибут `serializable`, службы обеспечения живучести библиотеки базового класса должны по умолчанию обеспечить живучесть для всех полей и свойств класса. В некоторых случаях поле или свойство не должно обладать живучестью. Это может быть вызвано соображениями повышения быстродействия или корректности (например, нет никакого смысла обеспечивать живучесть дескриптора файла). Атрибут `nonserialized`, помещенный перед объявлением поля или свойства, препятствует его сохранению в потоке.

serializable

```
[serializable]
```

Примечание

Атрибут `serializable` обеспечивает живучесть полей и свойств класса или структуры. Этот атрибут используется однократно. Его полное имя `System.Serialization.SerializableAttribute`.

Класс атрибута `serializable` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Delegate |
AttributeTargets.Enum | AttributeTargets.Struct)]
public class SerializableAttribute: System.Attribute
{
```

```
        public SerializableAttribute()
        {
                ...
        }
}
```

Базовая библиотека классов обеспечивает живучесть только тех классов и структур, которые имеют атрибут `serializable`. Для того чтобы сохранить поле, необходимо, чтобы тип этого поля обладал живучестью. Так, в приведенном ниже примере тот факт, что класс `Some_Class` обладает живучестью, недостаточен для сохранения его поля `Some_Struct`, если оно, в свою очередь, не обладает живучестью:

```
[serializable]
public class Some_Class
{
                                int i;
                                Some_Struct   sc;
[nonserialized]       int     temp; // Нет необходимости сохранять
}
[serializable] // Необходимо указать для сохранения
public struct Some_Struct
{
        string s;
}
```

obsolete

```
[obsolete(message <>)]
[obsolete(message <>, iserror <>)]
```

Аргументы

- `message` — имеет тип `string`. В идеальном случае — популярное объяснение того, почему этот элемент является устаревшим, и что следует использовать вместо него;
- `iserror` — имеет тип `bool`. Если данный аргумент имеет значение `true`, компилятор должен рассматривать использование данного элемента как ошибку. По умолчанию установлено значение `false` (компилятор выдает предупреждение).

Примечание

Атрибут `obsolete` может использоваться в любом объявлении, допускающем использование атрибутов для отметки компонентов программы, которые не должны использоваться. Данный атрибут используется однократно. Его полное имя `System.ObsoleteAttribute`.

При использовании в программе компонента, помеченного атрибутом `obsolete`, компилятор выдает предупреждение или сообщение об ошибке (в зависимости от значения аргумента `IsError`). В окне сообщения на экране появляется текст аргумента `message`.

Класс атрибута `obsolete` имеет следующее описание:

```
[AttributeUsage(AttributeTargets.All)]
public class ObsoleteAttribute: System.Attribute
{
        public ObsoleteAttribute(string message)
        {
                ...
        }
        public string Message
        {
                get
                {
                        ...
                }
        }
        public bool IsError
        {
                get
                {
                        ...
                }
                set
                {
                        ...
                }
        }
}
```

Ниже приведен пример использования данного атрибута:

```
using System;
public class MyClass
{
        [obsolete("Do not use OldWay; Use NewWay instead", true)]
        static void OldWay()
        {
                Console.WriteLine("Clinton");
        }
        static void NewWay()
        {
                Console.WriteLine("Bush");
        }
        public static void Main()
        {
                OldWay();
        }
}
```

Используемые перечислимые типы

В приведенных выше классах атрибутов используются аргументы перечислимых типов, созданных специально для работы с данными атрибутами. Некоторые из этих типов используются более чем в одном классе атрибутов. Поэтому описание этих типов вынесено в специальный раздел.

AttributeTargets

```
Public enum AttributeTargets
{
        Assembly      = 0x0001,
        Module        = 0x0002,
        Class         = 0x0004,
        Struct        = 0x0008,
        Enum          = 0x0010,
        Constructor   = 0x0020,
        Method        = 0x0040,
        Property      = 0x0080,
        Field         = 0x0100,
        Event         = 0x0200,
        Interface     = 0x0400,
```

```
        Parameter       = 0x0800,
        Delegate       = 0x1000,
        All            = Assembly | Module | Class | Struct | Enum |
Constructor | Method | Property | Field | Event | Interface | Parameter |
                         Delegate,
        ClassMembers   = Class | Struct | Enum | Constructor | Method |
                         Property | Field | Event | Delegate | Interface
}
```

Примечание

Этот перечислимый тип используется в классе атрибута `AttributeUsage`.

CallingConvention

```
public enum CallingConvention
{
        Winapi         = 1,
        Cdecl          = 2,
        StdCall        = 3,
        Thiscall       = 4,
        FastCall       = 5
}
```

Примечание

Этот перечислимый тип используется в классе атрибута `dllimport`.

CharSet

```
public enum CharSet
{
  None = 1, // Программист не задал способ маршалинга строк.
  Ansi  = 2, // Строки при маршалинге имеют стандарт ANSI: 1 бит на символ.
  Unicode = 3, // Строки при маршалинге имеют стандарт Unicode: 2
               // бита на символ.
  Auto = 4     // Маршалинг строк определяется операционной системой.
}
```

Примечание

Этот перечислимый тип используется в классах атрибутов `dllimport` и `structlayout`.

COMVisibility

```
public enum COMVisibility
{
        VisibilityDefault            = 0,
        sibilityOmitted              = 1
}
```

Примечание

Этот перечислимый тип используется в классе атрибута `comvisibility`.

ComInterfaceType

```
public enum ComInterfaceType
{
        Dual           = 0,
        IUnknown       = 1,
        IDispatch      = 2
}
```

Примечание

Этот перечислимый тип используется в классе атрибута `interfacetype`.

LayoutKind

```
public enum LayoutKind
{
        Sequential,
        Union,
        Explicit
}
```

Примечание

Этот перечислимый тип используется в классе атрибута `structlayout`.

UnmanagedType

```
public serializable enum UnmanagedType
{
        End                   = 0x0,
        Void                  = 0x1,
        Boolean               = 0x2,
```

```
        I1                  = 0x3,
        U1                  = 0x4,
        I2                  = 0x5,
        U2                  = 0x6,
        I4                  = 0x7,
        U4                  = 0x8,
        I8                  = 0x9,
        U8                  = 0xa,
        R4                  = 0xb,
        R8                  = 0xc,
        SysChar             = 0xd,
        Variant             = 0xe,
        Currency            = 0xf,
        Ptr                 = 0x10,
        Decimal             = 0x11,
        Date                = 0x12,
        BStr                = 0x13,
        LPStr               = 0x14,
        LPWStr              = 0x15,
        LPTStr              = 0x16,
        FixedSysString      = 0x17,
        ObjectRef           = 0x18,
        IUnknown            = 0x19,
        Idispatch           = 0x1a,
        Struct              = 0x1b,
        Interface           = 0x1c,
        SafeArray           = 0x1d,
        FixedArray          = 0x1e,
        Int                 = 0x1f,
        UInt                = 0x20,
        NestedStruct        = 0x21,
        ByValStr            = 0x22,
        AnsiBStr            = 0x23,
        TBStr               = 0x24, // В зависимости от платформы
                                    // выбирается BSTR или ANSIBSTR
        Max                 = 0x25  // Первый некорректный тип элемента
}
```

Примечание

Этот перечислимый тип используется в классе атрибута `marshal`.

Глава 13

Реализация пользовательского интерфейса

Как уже говорилось выше, в языке программирования C# используется ограниченный набор средств ввода и вывода информации, поскольку предполагается, что все эти операции будут выполняться в компонентах, написанных на других языках программирования. Однако это не исключает необходимости включения в данный язык средств для преобразования формата данных, для получения информации из объектов и передачи в них информации.

Работа со строками

При реализации пользовательского интерфейса часто приходится иметь дело со строками. Это связано с тем, что большая часть информации поступает от пользователя в текстовом виде и передается тоже в текстовом виде.

Строка в C# представляет собой объект класса `string`, полное имя которого — `System.String`. Этот класс имеет большое число методов, поскольку в C# запрещено непосредственное внесение изменений в строки, а все изменения в них могут производиться только с использованием соответствующих методов. Методы не изменяют объект, для которого они вызываются, а возвращают новый объект, в который внесены требуемые изменения.

Для сравнения строк и поиска в них образца могут быть использованы следующие методы:

- `Compare` — производит сравнение двух строк.
- `CompareOrdinal` — производит сравнение двух строк без учета их формата (включающего в себя признак национальной кодировки).
- `CompareTo` — производит сравнение данной строки с другой строкой.
- `EndsWith` — проверяет, совпадает ли указанный образец с завершением данной строки.
- `IndexOf` — возвращает индекс первого вхождения указанного образца в данную строку.

- `LastIndexOf` — возвращает индекс последнего вхождения указанного образца в данную строку.
- `StartsWith` — проверяет, совпадает ли указанный образец с началом данной строки.

Для модификации строк могут быть использованы следующие методы:

- `Concat` — объединяет два и более объектов в одну строку. Если объект не является строкой, для него вызывается метод `ToString`.
- `CopyTo` — копирует указанное количество символов из указанной позиции данной строки в указанную позицию массива символов.
- `Insert` — возвращает строку, получаемую путем вставки образца в указанную позицию данной строки.
- `Join` — объединяет элементы массива строк в одну строку, помещая между ними разделители.
- `PadLeft` — выравнивает строку по правому краю, помещая слева символы пробела или указанные символы для обеспечения указанной длины.
- `PadRight` — выравнивает строку по левому краю, помещая справа символы пробела или указанные символы для обеспечения указанной длины.
- `Remove` — уничтожает указанное количество символов, начиная с указанной позиции.
- `Replace` — заменяет все вхождения одного символа в строке другим символом.
- `Split` — создает из данной строки массив строк, используя в качестве разделителя указанный символ или массив символов.
- `Substring` — возвращает подстроку данной строки, начинающуюся с указанной позиции и, в зависимости от перегрузки функции, заканчивающуюся концом данной строки или содержащую указанное количество символов.
- `ToLower` — преобразует все символы строки в нижний регистр.
- `ToUpper` — преобразует все символы строки в верхний регистр.
- `Trim` — удаляет из начала и конца строки все указанные символы (по умолчанию удаляются пробелы).
- `TrimEnd` — удаляет из конца строки все указанные символы (рекурсивное удаление последнего символа строки прекращается, когда будет встречен символ, не входящий в указанный набор).
- `TrimStart` — удаляет из начала строки все указанные символы (рекурсивное удаление первого символа строки прекращается, когда будет встречен символ, не входящий в указанный набор).

Форматирование

При выводе числового значения его, как правило, необходимо сначала преобразовать в текстовый вид. При работе со стандартными типами для этого может быть использован метод `Format` данного типа. Этот же метод неявно вызывается при обращении к методам `Console.Write` и `Console.WriteLine`.

Метод `Format` имеет несколько перегрузок. Первым аргументом одной из перегрузок является строка форматирования, определяющая формат, в котором должно выводиться соответствующее значение. Если данный метод не имеет аргументов, значение выводится в формате, используемом по умолчанию.

Стандартные строки форматирования

Стандартная строка форматирования состоит из символа, определяющего формат, и последовательности цифр, определяющих размерность полей этого формата. Рассмотрим основные форматы.

Целочисленный десятичный формат

Этот формат используется для вывода целых чисел в десятичном формате. Ему соответствуют символы `d` и `D`. За этими символами может размещаться только одно числовое значение, определяющее минимальное число десятичных символов в результате преобразования. Если результат преобразования содержит меньшее число десятичных символов, недостающие позиции заполняются нулями.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
                int    i = 123;

                Console.WriteLine( i );
                Console.WriteLine( "{0:d5}", i );
                Console.WriteLine( "{0:D2}", i );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
123                     00123                   123
```

Целочисленный шестнадцатеричный формат

Этот формат используется для вывода целых чисел в шестнадцатеричном формате. Ему соответствуют символы `x` и `X`. За этими символами может размещаться только одно числовое значение, определяющее минимальное число шестнадцатеричных символов в результате преобразования. Если результат преобразования содержит меньшее число шестнадцатеричных символов, недостающие позиции заполняются нулями. К сожалению, при выводе значения в него не включается информация о формате вывода (`0x`), что в некоторых случаях может привести к неправильной трактовке выведенных значений.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
                int    i = 123;

                Console.WriteLine( "{0:x}", i );
                Console.WriteLine( "{0:x5}", i );
                Console.WriteLine( "{0:X1}", i );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
7B                      0007B                   7B
```

Формат с фиксированной запятой

Этот формат используется для вывода чисел с плавающей точкой. Ему соответствуют символы `f` и `F`. За этими символами может размещаться одно числовое значение, определяющее число знаков, выводимое после запятой.

Рассмотрим это на примере:

```
using System;

class CDemo
{
```

```
        public static void Main()
        {
                double d = 123.456;

                Console.WriteLine( d );
                Console.WriteLine( "{0:f}", d );
                Console.WriteLine( "{0:f5}", d );
                Console.WriteLine( "{0:F0}", d );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
123,456                         123,46
123,45600                       123
```

Обратите внимание на то, что при выводе в этом формате без указания точности число округляется, хотя точность его представления этого не требует.

Экспоненциальный формат

Этот формат используется для вывода чисел с плавающей точкой, имеющих большой диапазон значений. Этот формат также иногда называется научным форматом, в соответствии с обычной областью его использования. Число, выводимое в данном формате, разбивается на мантиссу и порядок. Мантисса представляет собой результат приведения абсолютного значения числа в диапазон от 1 до 10 путем деления или умножения его на целую степень числа 10. Эта степень и ее знак помещаются в порядок числа.

Данному формату соответствует символы `e` и `E`. За этими символами может размещаться одно числовое значение, определяющее число знаков, выводимое после запятой в мантиссе числа. По умолчанию мантисса представляется с точностью шести знаков после запятой.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
                double d = 123.456;
                double e = .000789;

                Console.WriteLine( "{0:e}", d );
```

```
            Console.WriteLine( "{0:e3}", d );
            Console.WriteLine( "{0:E}", e );
            Console.WriteLine( "{0:E2}", e );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
1,234560E+002                    1,235E+002
7,890000E-004                    7,89E-004
```

Общий формат

Этот формат используется для вывода чисел с плавающей точкой. В зависимости от диапазона выводимого значения в нем используется формат с фиксированной запятой или экспоненциальный формат (тот, который обеспечит более компактную запись). Данному формату соответствуют символы `g` и `G`. За этими символами может размещаться одно числовое значение, назначение которого понять очень сложно.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
            double d = 123456.789;
            double e = .0009876543;

            Console.WriteLine( "{0:g}", d );
            Console.WriteLine( "{0:g3}", d );
            Console.WriteLine( "{0:G3}", e );
            Console.WriteLine( "{0:G16}", e );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
123456,789                       1,23E5
0,000988                         0,0009876543
```

Формат с выделением тысяч

Этот формат используется для вывода чисел с плавающей запятой. При выводе этих чисел их целая часть разбивается на триады, которые разделяются пробелом. При выводе в этом формате целых чисел они преобразуются в формат чисел с плавающей точкой.

Примечание

Используемый разделитель разрядов определяется используемым объектом класса `NumberFormatInfo`.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
                double d = 123456.789;
                int    i = 123456789;

                Console.WriteLine( "{0:n}", d );
                Console.WriteLine( "{0:n3}", d );
                Console.WriteLine( "{0:N3}", i );
                Console.WriteLine( "{0:N}", i );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
123 456,79                          123 456,789
123 456 789,000                     123 456 789,00
```

Финансовый формат

Этот формат используется для вывода чисел с плавающей точкой. Выводимые в этом формате значения представляются в виде, определяемом пользовательской настройкой Windows. При выводе в этом формате целых чисел они преобразуются в формат чисел с плавающей точкой.

Рассмотрим это на примере:

```
using System;

class CDemo
{
```

```
        public static void Main()
        {
                double d = 123456.789;
                int           i = 123456789;

                Console.WriteLine( "{0:c}", d );
                Console.WriteLine( "{0:c3}", -d );
                Console.WriteLine( "{0:C3}", i );
                Console.WriteLine( "{0:C}", -i );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
123 456,79р.                          -123 456,789р.
123 456 789,000р.                     -123 456 789,00р.
```

Как видно, этот формат отличается от формата с разделением на тысячи только наличием знака денежной единицы, в данном случае рублей в конце значения.

Модификация формата вывода

Стандартные форматы вывода значений не во всех случаях могут удовлетворить потребности разработчиков программного обеспечения. Иногда возникает необходимость внести в форматы некоторые коррективы. Для этого используются специальные нестандартные символы форматирования. Все остальные символы, кроме стандартных и нестандартных символов форматов, непосредственно копируются в результирующую строку.

Форматирование целых чисел

Для вставки символов в выводимые целые числа в нестандартной строке форматирования для каждой позиции результирующей строки определяется, что именно будет в ней выводиться. Если в данной позиции стоит символ `0`, то в ней будет выводиться символ преобразованной строки или нулевое значение, если данная позиция результирующей строки не используется. Если же в этой позиции стоит символ `#` (фунт) — то эта позиция будет игнорироваться, если до нее в строке форматирования не стоит символ `0` или любой другой символ. Если стоит символ `0`, то в этой позиции будет выводиться нулевое значение или символ преобразованной строки. Если же это другой символ, то данная позиция будет опускаться или же содержать символ преобразованной строки.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
                int            i = 1234;

                Console.WriteLine( "{0:#0#####}", i );
                Console.WriteLine( "{0:## in ###00}", i );
                Console.WriteLine( "{0:0### in 00}", i );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
001234                  in 1234                  0012 in 34
```

К сожалению, этот метод не может быть использован для вывода в шестнадцатеричном формате.

Форматирование чисел с плавающей точкой

Для вставки символов в выводимые целые числа в нестандартной строке форматирования для каждой позиции результирующей строки определяется, что именно будет в ней выводиться. Если в данной позиции стоит символ `0`, то в ней будет выводиться символ преобразованной строки или нулевое значение, если данная позиция результирующей строки не используется. Если же в этой позиции стоит символ `#` (фунт) — то эта позиция будет игнорироваться, если до нее в строке форматирования не стоит символ `0` или любой другой символ. Если стоит символ `0`, то в этой позиции будет выводиться нулевое значение или символ преобразованной строки. Если же это другой символ, то данная позиция будет опускаться или же содержать символ преобразованной строки.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
```

```
            double        d = 1234.56789;

            Console.WriteLine( "{0:#0####.#######}", d );
            Console.WriteLine( "{0:######.#####0#}", d );
            Console.WriteLine( "{0:## in ##.00}", d );
            Console.WriteLine( "{0:0##. in 00######}", d );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
01234,56789                          1234,567890
12 in 34,57                          1234 in ,56789
```

Обратите внимание на то, что в последнем примере запятая выводится после текста, хотя в программе она стоит до него.

Выделение тысяч

Использование нестандартных строк форматирования не отменяет требований к удобству вывода информации. Поэтому в них, так же как и в стандартных строках форматирования предусмотрена возможность разбиения числа на триады. Для этого используется символ запятой.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
            double        d = 1234.56789;
            int           i = 1234567;

            Console.WriteLine( "{0:#0####,###.#######}", d );
            Console.WriteLine( "{0:#0##,#,#,#,##.#######}", d );
            Console.WriteLine( "{0:#0#######.##,#####}", d );
            Console.WriteLine( "{0:######.#####0#}", d );
            Console.WriteLine( "{0:## in #,#.00}", d );
            Console.WriteLine( "{0:0#,#. in 00##,####}", d );
            Console.WriteLine( "{0:#,#}", i );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
00 001 234,56789                    00 001 234,56789
00001234,56789                      1234,567890
1 2 in 34,57                        1 234 in ,56789
1 234 567
```

Как видно из приведенного примера, количество запятых и их расположение при условии, что хотя бы одна из них расположена слева от десятичной точки, никак не влияют на разбиение строки результата. При разбиении не учитываются другие символы. Этот способ позволяет разбивать на триады целые числа.

Масштабирование

В языке C# изначально была предусмотрена возможность форматирования выводимой величины. Для этого предполагалось использовать символ запятой, размещаемый сразу же после десятичной точки. Каждый такой символ должен был уменьшать выводимое значение на 1000. Однако в Visual Studio.NET такая возможность не была реализована и неизвестно, будет ли она реализована в дальнейшем.

Вывод процентов

В нестандартной строке программирования предусмотрена возможность вывода процентов числа. При помещении в нее символа `%` выводимое значение увеличивается в 100 раз, и данный символ выводится в своей позиции. При помещении в эту строку нескольких символов процентов, каждый из них приведет к увеличению выводимого значения и будет отображен в своей позиции.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
                double         d = 12.456789;

                Console.WriteLine( "{0:######.######%}", d );
                Console.WriteLine( "{0:###%###.#%#####}", d );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
12435,6789%                          124%567,8%9
```

Изменение формата чисел

В некоторых случаях возникает необходимость задания различного формата для вывода положительных и отрицательных чисел. Для разделения этих форматов в нестандартной строке форматирования используется символ ;. Если строка форматирования состоит только из двух секций, то первая из них определяет формат положительных и нулевых значений, а вторая — отрицательных значений. Если строка форматирования состоит из трех секций, то первая из них определяет формат положительных значений, вторая — отрицательных значений, а третья — нулевых значений.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
                double        d = 12.456789;

                Console.WriteLine( "{0:0.######;0.#}", d );
                Console.WriteLine( "{0:0.######;0.#}", 0.0 );
                Console.WriteLine( "{0:0.######;0.#}", -d );
                Console.WriteLine( "{0:0.#;(0.0);0.00000}", d );
                Console.WriteLine( "{0:0.#;(0.0);0.00000}", 0 );
                Console.WriteLine( "{0:0.#;(0.0);0.00000}", -d );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
12,456789                            0
12,5                                 12.5
0.00000                              (12.5)
```

Включение литералов

При выводе информации может возникнуть необходимость включить в нее символы, объявленные служебными в нестандартных строках форматирования. Для указания того, что данный символ не является служебным, перед

ним ставится символ обратной косой черты. Поскольку этот же символ используется для задания литералов C#, то он будет встречаться дважды. Для того чтобы избежать этого, перед началом строки форматирования можно установить символ дословного синтаксиса `@`. При выводе последовательности из нескольких служебных символов их можно заключить в апострофы.

Рассмотрим это на примере:

```
using System;

class CDemo
{
        public static void Main()
        {
                int            i = 12456789;

                Console.WriteLine( "{0:###\\####}", i );
                Console.WriteLine( @"{0:###\0###}", i );
                Console.WriteLine( "{0:######'%#0'}", i );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
12456#789                124560789                12456789%#0
```

Форматирование объектов

Для определения формата вывода объекта используется интерфейс `IFormattable` и его единственный метод `Format`. Этот интерфейс должен быть реализован в классе или структуре, объект которой будет выводиться, и в нем должны быть реализованы все операции по выводу информации в соответствии с установленными ключами.

Рассмотрим это на примере:

```
using System;

class MyClass: IFormattable
{
        public MyClass( int i, string sz )
        {
                this.i = i;
                this.sz= sz;
        }
```

```
        public string Format(string format, IServiceObjectProvider sop)

        {
                if( (format != null) && (format.Equals("S") ||
                     format.Equals("s")) )
                {
                        return String.Format("{0}: {1}", sz, i );
                }
                else
                        return i.Format( format, sop );
        }

        int             i;
        string sz;
}

class CDemo
{
        public static void Main()
        {
                MyClassmc     = new MyClass( 5, "Value" );

                Console.WriteLine( "With Format:\n   {0:s}", mc );
                Console.WriteLine( "Without Format:\n   {0}", mc );

                Console.ReadLine();
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
With Format:
        Value: 5
Without Format:
        5
```

Внесение изменений в существующие форматы

В некоторых случаях возникает необходимость кардинальным образом изменить существующие форматы отображения значений соответствующего типа. Для этого используются интерфейс `IServiceObjectProvider` и его метод `GetServiceObject` и интерфейс `ICustomFormatter` и его метод `Format`.

Переопределение этих методов в классе, производящем форматирование целых чисел в шестнадцатеричном формате, продемонстрировано в следующем примере:

```
using System;

class HexFormat: IServiceObjectProvider, ICustomFormatter
{
        public object GetServiceObject( Type service )
        {
                if( service == typeof(ICustomFormatter) )
                {
                        return this;
                }
                else
                        return null;
        }

        public string Format(string format, object obj,
                             IServiceObjectProvider sop)

        {
                if( format == null )
                        return String.Format("{0}", obj );

                if( (format.Equals("x") || format.Equals("X")) )
                        return String.Format(@"0x{0:x1}", obj );
                else
                        return String.Format("{0:" + format + "}", obj );
        }
}

class CDemo
{
        public static void Main()
        {
                HexFormat     hf     = new HexFormat();
                int                  i      = 1234;

                Console.WriteLine( String.Format("Hex Format: {0:x}", new
                                   object[] {i}, hf));
```

```
            Console.WriteLine( String.Format("Dec Format: {0:d}", new
                               object[] {i}, hf));


            Console.ReadLine();
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
Hex Format: 0x4d2
Dec Format: 1234
```

Преобразование строк в значения

Преобразование строк, содержащих числовую информацию, в соответствующие им значения производится методом `Parse` того типа, к которому производится преобразование. При этом производится проверка допустимости такого преобразования. Если это преобразование недопустимо, вызывается исключение `System.FormatException`:

```
using System;

class CDemo
{
        public static void Main()
        {
            int           i = Int32.Parse("1234");
            double d = Double.Parse("12,34");

            Console.WriteLine( i );
            Console.WriteLine( d );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
1234                                    12.34
```

Следует обратить внимание на то, что в преобразуемой строке вещественного формата для отделения целой части от дробной используется запятая, а в выводимой строке — десятичная точка. Если поставить в преобразуемой строке десятичную точку, в процессе выполнения программы будет вызвано исключение. То же самое произойдет и при попытке преобразования строки вещественного формата к целочисленной величине.

Чтение и запись файлов

Средства для работы с файлами в среде CLR принадлежат пространству имен `System.IO`. Эти средства включают в себя набор классов, производных от класса `Stream`, реализующих основные операции ввода/вывода для различных устройств. Ниже приведен список основных классов;

- `BinaryReader` — используется для чтения двоичных (неструктурированных) данных.
- `BinaryWriter` — используется для записи двоичных (неструктурированных) данных.
- `BufferedStream` —используется для буферизации данных при совместной работе нескольких потоков. Создаваемый данным классом поток реализуется на базе другого потока.
- `FileStream` — реализует основные операции чтения и записи информации в файл.
- `MemoryStream` — реализует основные операции чтения и записи информации в файл, расположенный в оперативной памяти.
- `NetworkStream` — реализует основные операции чтения и записи информации в сетевой файл.
- `Stream` — базовый (абстрактный) класс для всех классов потокового ввода/вывода. Содержит методы для синхронного и асинхронного чтения и записи данных на уровне байтов.
- `StreamReader` — используется для чтения символов в указанной кодировке из файла.
- `StreamWriter` — используется для записи символов в указанной кодировке в файл.
- `StringReader` — используется для чтения строк из файла.
- `StringWriter` — используется для записи строк в файл.
- `TextReader` — базовый (абстрактный) класс для всех классов потокового чтения текстовой информации.
- `TextWriter` — базовый (абстрактный) класс для всех классов потоковой записи текстовой информации.
- `XmlTextReader` — аналогичен классу `TextReader`, но используется для декодирования XML-кода.
- `XmlTextWriter` — аналогичен классу `TextWriter`, но используется для создания XML-кода.

Ниже приведен пример использования перечисленных выше классов для записи текстовой информации в файл:

```
using  System;
using  System.IO;

class CDemo
{

       public static void Main()
       {
              FileStream    file   = new FileStream( "Demo.txt",
                                     FileMode.Create );
              StreamWriter  sw     = new StreamWriter( file );

              sw.WriteLine("It is only a demo");

              sw.Close();
              file.Close();
       }
}
```

Автоматическое сохранение объектов

Программисты на Visual C++, знакомые с концепцией "документ/представление", уже привыкли к тому, что документ сам должен заботиться о сохранении своего содержимого. То же самое относится и к входящим в него объектам. Этот процесс иногда называют сериализацией (serialization). В среде CLR для этого применяются метаданные объекта, которые должен создать его разработчик. Для этого используются атрибуты `serializable` и `nonserialized`. Первый из них разрешает автоматическое сохранение объекта, а второй запрещает сохранение значений отдельных его членов:

```
using  System;
using  System.IO;
using  System.Collections;
using  System.Runtime.Serialization;
using  System.Runtime.Serialization.Formatters.Binary;

[serializable]
class MyClass: IFormattable
{
```

```
        public MyClass( int i, string sz )
        {
                this.i = i;
                this.sz= sz;
        }
        public string Format(string format, IServiceObjectProvider sop)

        {
         if( (format != null) && (format.Equals("S") ||
         format.Equals("s")) )
                {
                        return String.Format("{0}: {1}", sz, i );
                }
                else
                        return i.Format( format, sop );
        }

        [nonserialized]
        int             i;      // Значение поля не сохраняется
        string sz;
}

class CDemo
{
        public static void Main()
        {
                MyClassmc     = new MyClass( 5, "Value" );

                // Вывод исходного состояния

                Console.WriteLine( "Initial State:\n   {0:s}", mc );

                // Сохранение в файле

                Stream sw = File.Create("Demo.bin");
                BinaryFormatter bw = new BinaryFormatter();
                bw.Serialize( sw, mc );
                sw.Close();

                // Чтение из файла

                Stream sr = File.OpenRead("Demo.bin");
```

```
            BinaryFormatter br = new BinaryFormatter();
            MyClassmcn = (MyClass) br.Deserialize( sr );
            sr.Close();

            // Вывод сохраненной информации

            Console.WriteLine( "Saved State:\n   {0:s}", mcn );
        }
}
```

В результате выполнения этой программы на экран будут выведены следующие строки:

```
Initial State:
       Value: 5
Saved State:
       Value: 0
```

Как видно из приведенного текста, сохранилось только значение строкового поля, а значение целочисленного поля было утеряно.

Работа с потоками

Распределенные приложения, для создания которых и был разработан язык C#, по своей сути являются многозадачными, то есть включают в себя несколько независимых процессов, каждый из которых может одновременно исполняться в различных потоках. До сих пор все создаваемые приложения исполнялись в одном потоке, и у читателя могло закрасться подозрение, что C# не может работать с потоками. Спешу это подозрение развеять. Для работы с потоками в CLR определено специальное пространство имен `System.Threading`, содержащее классы, используемые для работы с потоками.

Для вызова рабочего метода потока используется делегат `ThreadStart`, объект которого передается в качестве аргумента конструктору класса потока. Делегат `ThreadStart` имеет следующий синтаксис:

```
public delegate void ThreadStart();
```

То есть в качестве рабочей функции потока может использоваться любой метод, не имеющий формальных параметров и не возвращающий значения.

Для запуска потока используется метод `Thread.Start`, вызывающий делегат `ThreadStart` и создающий для его выполнения отдельный поток. Работа потока завершается с выходом из его рабочей функции или при вызове в вызывающем потоке метода `Thread.Stop`. При возникновении необходимости приостановить работу потока на определенный промежуток времени вызывается метод `Thread.Sleep`.

Одновременное исполнение процесса в различных потоках ставит задачу их синхронизации. В C# для этого используется оператор `lock`. Работа этого оператора аналогична работе функций Win32, используемых для работы с критическими секциями. Рассмотрим работу этого оператора на примере:

```
using System;
using System.Threading;

public class  MyThread
{
       public void StingsOut()
       {
              int    i;

              for(i=0; i < 10; i++)
              {
                     Console.Write("This");
                     Thread.Sleep(1);
                     Console.Write(" is");
                     Thread.Sleep(1);
                     Console.Write(" a very");
                     Thread.Sleep(1);
                     Console.Write(" long");
                     Thread.Sleep(1);
                     Console.Write(" string");
                     Thread.Sleep(1);
                     Console.Write(" number ");
                     Thread.Sleep(1);
                     Console.WriteLine( i );
              }
       }
}

public class Test
{
       public static void Main()
       {
              MyThread      mt     = new MyThread();
              Thread First  = new Thread(new ThreadStart(mt.StingsOut));
```

```
            Thread        Second = new Thread(new Thread-
Start(mt.StingsOut));

            First.Start();
            Second.Start();
        }
}
```

В приведенном примере создается объект класса `MyThread`, и в различных потоках вызывается его метод, выводящий длинную строку. Вывод строки организован таким образом, чтобы между выводом отдельных ее фрагментов существовали пуазы, в которые мог бы вклиниться другой поток. Поэтому в результате выполнения этого приложения строки, выводимые этими потоками, перемешиваются. Чтобы не допустить этого, изменим программу следующим образом:

```
using System;
using System.Threading;

public class  MyThread
{
        public void StingsOut()
        {
            int    i;

            for(i=0; i < 10; i++)
            {
                lock(this)
                {
                    Console.Write("This");
                    Thread.Sleep(1);
                    Console.Write(" is");
                    Thread.Sleep(1);
                    Console.Write(" a very");
                    Thread.Sleep(1);
                    Console.Write(" long");
                    Thread.Sleep(1);
                    Console.Write(" string");
                    Thread.Sleep(1);
                    Console.Write(" number ");
                    Thread.Sleep(1);
```

```
                                Console.WriteLine( i );
                        }
                }
        }
}

public class Test
{
        public static void Main()
        {
                MyThread      mt     = new MyThread();
                Thread First  = new Thread(new ThreadStart(mt.StingsOut));
                Thread Second = new Thread(new ThreadStart(mt.StingsOut));

                First.Start();
                Second.Start();
        }
}
```

В этой программе все операторы, выводящие одну строку, помещены в блок оператора `lock`, предотвращающего доступ к данному объекту класса из другого потока на время вывода строки. Если бы в приведенном примере для создания потоков использовались два различных объекта класса `MyThread`, то использование оператора `lock` не исправило бы ситуацию, поскольку в этом случае отсутствовал бы разделяемый ресурс.

Глава 14

Включение отладочной информации

При создании приложений часто возникает необходимость обусловить вызов какой-либо функции. Чаще всего условием является задание режима отладки приложения, однако, могут быть заданы и другие условия, определяемые потребностями разработчика. Сразу же следует отметить, что этот механизм работает не только в отладочной, но и в распространяемых версиях.

Условный вызов методов

В C++ задача создания функций, вызываемых только в отладочной версии приложения, решалась с помощью макросов, включавшихся как в файл заголовка, так и в файл реализации класса. В C# для этого используются атрибуты, точнее, единственный атрибут `conditional`. Пример использования этого атрибута приведен ниже:

```
using  System;
using  System.Diagnostics;

class  DebugClass
{
       [conditional("DEBUG")]
       public void Report()
       {
              Console.WriteLine("This is a debug version");
       }
}

class Demo
{
       public static void Main()
       {
              DebugClass dc = new DebugClass();
```

```
            dc.Report();

            Console.WriteLine("Program is terminated");
        }
}
```

Поскольку по умолчанию среда программирования Visual Studio.NET создает отладочную версию приложения, при запуске этой программы на исполнение на экран будет выведен следующий текст:

```
This is a debug version
Program is terminated
```

Чтобы создать распространяемую версию приложения выберите команду меню **Build**, **Configuration Manager...**. Появится диалоговое окно **Configuration Manager**, изображенное на рис. 14.1.

Рис. 14.1. Диалоговое окно **Configuration Manager**

Раскройте список в столбце **Configuration** строки вашего проекта и выделите в нем строку **Release**. Эта строка появится в текстовом поле раскрывающегося списка. Закройте диалоговое окно **Configuration Manager** и запустите приложение на исполнение. На экран будет выведен текст:

```
Program is terminated
```

То есть метод `DebugClass.Report` вызываться не будет.

Классы Debug и Trace

Условный вызов функций позволяет разработчику включать в приложение собственные отладочные функции, однако, в CLR включены и стандартные классы, используемые для отладки приложений. Это классы `Debug` и `Trace`, принадлежащие пространству имен `System.Diagnostics`. Эти классы во

многом похожи, но используются в разных ситуациях. Основное отличие между ними заключается в том, что класс `Debug` не включается в распространяемую версию приложения, и все обращения к нему игнорируются, в то время как класс `Trace` включается в распространяемую версию приложения, что позволяет использовать его для отладки этой версии.

Включение в приложение класса `Debug` зависит от определения символьной переменной `DEBUG`, а включение класса `Trace` — от определения символьной переменной `TRACE`.

Проверка условий

Для проверки условий в классах `Debug` и `Trace` используется метод `Assert`. Синтаксис перегрузок этого метода в обоих классах одинаков. Ниже приведен пример использования одной из перегрузок этого метода в отладочной версии приложения:

```
class  Test
{
       public Test( int i )
       {
              this.i = i;
       }

       [conditional("DEBUG")]
       public void Report()
       {
              Debug.Assert( i <= 0, "Member must be greater then 0");
       }

       int i;
}

class Demo
{
       public static void Main()
       {
         Debug.Listeners.Clear();
         Debug.Listeners.Add( new TextWriterTraceListener(Console.Out));

              Test First    = new Test( 1 );
```

```
            Test Second   = new Test( 0 );
            First.Report();
            Second.Report();      // Будет выведено сообщение об ошибке
        }
}
```

По умолчанию вся отладочная информация выводится всем зарегистрированным элементам коллекции `Debug.Listeners`, имеющей тип `TraceListeners`. Этот член присутствует и в классе `Trace`. По умолчанию эта коллекция содержит единственный элемент — объект класса `DefaultTraceListeners`, осуществляющего вывод сообщения во всплывающее окно. Чтобы отменить этот вывод, коллекция очищается методом `TraceListeners.Clear`, а затем с использованием метода `TraceListeners.Add` в нее добавляется новый член, осуществляющий вывод в окно консольного приложения.

Поскольку в этом приложении ошибка обнаруживается только в одном из двух вызовов метода `DebugClass.Report`, то в результате его выполнения на экран будет выведен следующий текст:

```
Fail: Member must be greater then 0
```

Чтобы приложение могло вывести текст в окнах, определяемых элементами коллекции `Listeners`, в классах `Debug` и `Trace` определены методы `Write` и `WriteLine`, которые во всем остальном аналогичны одноименным методам класса `Console`.

В некоторых случаях разработчику приходится ограничивать объем выводимой на экран информации, производя ее фильтрацию по некоторому признаку. Это связано с тем, что достаточно большое приложение содержит значительное число точек, в которых может потребоваться вывести некоторую информацию. Как правило, можно заранее предугадать, в каких случаях эта информация может потребоваться. Поэтому ее вывод в остальных случаях приводит к засорению журнала отладки ненужной информацией.

Для управления объемом выводимой информации в классах `Debug` и `Trace` определены методы `WriteIf` и `WriteLineIf`. Эти методы аналогичны методам `Write` и `WriteLine` этих классов, однако, они выводят информацию только в том случае, если их первый аргумент имеет значение `true`. Рассмотрим пример использования этих функций:

```
using  System;
using  System.Diagnostics;

class  Test
{
```

```
        public Test( int i )
        {
                this.i = i;
        }

        [conditional("DEBUG")]
        public void Report()
        {
                Debug.WriteLineIf( doOut && (i <= 0), "Member must be
                                   greater then 0");
        }

        int            i;
        static public bool   doOut;
}

class Demo
{
        public static void Main()
        {
              Debug.Listeners.Clear();
              Debug.Listeners.Add( new TextWriterTraceListener(Console.Out));

              Test First      = new Test( 1 );
              Test Second     = new Test( 0 );

              First.Report();
              Second.Report();

              Test.doOut      = true;

              First.Report();
              Second.Report();
        }
}
```

В результате выполнения этой программы на экран будет выведен следующий текст:

```
Member must be greater then 0
```

Использование ключей для фильтрации выводимой информации

В предыдущем разделе была продемонстрирована возможность управления объемом выводимой информации с использованием логических переменных. Однако данный подход имеет один существенный недостаток: для изменения условий вывода информации необходимо вносить изменения в программу, что вообще-то считается дурным тоном в программировании. Лучше бы использовать для этой цели задаваемые извне ключи. Для решения такой задачи используется класс `BooleanSwitch`.

Перепишем предыдущий пример с использованием объекта класса `BooleanSwitch`:

```
using  System;
using  System.Diagnostics;

class  Test
{
       public Test( int i )
       {
              this.i = i;
       }

       [conditional("DEBUG")]
       public void Report()
       {
         Debug.WriteLineIf( doOut.Enabled && (i <= 0), "Member must be
                            greater then 0");
       }

       int            i;
       BooleanSwitch doOut  = new BooleanSwitch("doOutTest","Control de
                                                bug output");
}

class Demo
{
       public static void Main()
       {
         Debug.Listeners.Clear();
         Debug.Listeners.Add( new TextWriterTraceListener(Console.Out));
```

```
            Test First    = new Test( 1 );
            Test Second   = new Test( 0 );

            First.Report();
            Second.Report();

            Console.WriteLine("Program is terminated");
        }
}
```

Поскольку значение ключа `doOutTest` нигде не определено, при запуске этой программы на исполнение будет выведена строка:

```
Program is terminated
```

Существует несколько способов задания значения ключа. Здесь будет рассмотрено только использование для этого системного реестра. В этом случае для регистрации ключа и задания его значения необходимо создать в нем переменную, имеющую формат двойного слова (`DWORD`) и идентификатор `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\COMPlus\Switches\doOutTest`. Если в системном реестре будет отсутствовать ключ `Switches`, его следует создать. Этой переменной может быть присвоено значение 0 или 1. После присваивания ей значения 1 и запуска приложения на исполнение на экране появится следующий текст:

```
Member must be greater then 0
Program is terminated
```

Уровни трассировки

В процессе отладки сложной программы бывает полезно задать уровень детализации выводимых сообщений, то есть обращать внимание только на грубые ошибки и игнорировать предупреждения. Это позволит существенно сократить объем выводимой информации и охватить возникшую проблему в целом.

Для этого в классе `TraceSwitch` предусмотрена возможность выбора одного из пяти уровней трассировки, перечисленных в табл. 14.1.

Таблица 14.1. *Уровни трассировки*

Уровень	Числовое значение	Описание
`Off`	0	Трассировка отключена
`Error`	1	Выводятся только сообщения об ошибках
`Warning`	2	Выводятся и предупреждения

Таблица 14.1 *(окончание)*

Уровень	Числовое значение	Описание
Info	3	Выводится дополнительная информация
Verbose	4	Выводится вся информация

Каждый последующий уровень включает в себя и все предыдущие. Поэтому нельзя задать режим вывода предупреждений и запретить вывод сообщений об ошибках. Указанные числовые значения используются в переменных окружения и ключах системного реестра.

Для задания режима, в котором выводится данная информация, в классе `TraceSwitch` используются специальные свойства, имя которых образуется из слова `Trace` и имени соответствующего уровня. Пример использования этих свойств приведен ниже:

```
using  System;
using  System.Diagnostics;

class  Test
{
       public Test( int i )
       {
              this.i = i;
       }

       [conditional("DEBUG")]
       public void Report()
       {
              Debug.WriteLineIf( doOut.TraceInfo, "We are in a method");
              Debug.WriteLineIf( doOut.TraceError && (i <= 0), "Member
                                 must be greater then 0");
       }

       int            i;
       TraceSwitch    doOut  = new TraceSwitch("doOutSwitch","Control de
                                                bug output");
}

class Demo
{
       public static void Main()
       {
```

```
        Debug.Listeners.Clear();
        Debug.Listeners.Add( new TextWriterTraceListener(Console.Out));

            Test First    = new Test( 1 );
            Test Second   = new Test( 0 );

            First.Report();
            Second.Report();

            Console.WriteLine("Program is terminated");
        }
}
```

Как и в предыдущем примере, при первом запуске этой программы на экран выводится только сообщение о завершении программы, поскольку переменная, позволяющая задавать уровень трассировки, нигде не определена. Если же открыть системный реестр, поместить в него идентификатор `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\COMPlus\Switches\doOutSwitch`, имеющий формат двойного слова, и записать в него соответствующее значение, то информация, выводимая на экран приложением, изменится в зависимости от этого значения.

Использование пользовательских переключателей

Некоторым разработчикам может показаться, что пяти уровней трассировки им недостаточно. Единственное, что можно им посоветовать, это создать свой собственный класс для чтения ключей из системного реестра. Для этого нужно просто создать пользовательский класс, производный от класса `Switch`, включить в него пользовательские свойства и перегрузить его конструктор. Пример создания и использования такого пользовательского класса приведен ниже:

```
using  System;
using  System.Diagnostics;

class  CustomSwitch: Switch
{
       public CustomSwitch( string displayName, string description ):
base( displayName, description )
       {
       }
```

```
public bool Tiny
{
        get
        {
                return true;
        }
}

public bool Small
{
        get
        {
                return (SwitchSetting >= 1);
        }
}

public bool Normal
{
        get
        {
                return (SwitchSetting >= 2);
        }
}

public bool Big
{
        get
        {
                return (SwitchSetting >= 3);
        }
}

public bool Huge
{
        get
        {
                return (SwitchSetting >= 4);
        }
}
```

```
        public bool Enormous
        {
                get
                {
                        return (SwitchSetting >= 5);
                }
        }
}

class  Test
{
        public Test( int i )
        {
                this.i = i;
        }

        [conditional("DEBUG")]
        public void Report()
        {
                Debug.WriteLineIf( doOut.Huge, "We are in a method");
                Debug.WriteLineIf( doOut.Normal && (i <= 0), "Member must
                                   be greater then 0");
        }

        int            i;
        CustomSwitch  doOut  = new CustomSwitch("doOutSwitch","Control
                                                debug output");
}

class Demo
{
        public static void Main()
        {
                Debug.Listeners.Clear();
                Debug.Listeners.Add( new TextWriterTraceListener
                                   (Console.Out));

                Test First    = new Test( 1 );
                Test Second   = new Test( 0 );

                First.Report();
```

```
            Second.Report();

            Console.WriteLine("Program is terminated");

            Console.ReadLine();
        }
}
```

В свойствах методов класса `CustomSwitch` проверяется значение унаследованного от базового класса свойства `SwitchSetting`, используемого для доступа к текущему значению ключа.

Глава 15

Обработка исключений

В соответствии с современным американским (и не только) законодательством программ без ошибок не бывает. Это правило позволяет разработчикам программного обеспечения успешно противостоять натиску разъяренных пользователей, обнаруживающих в их программах все новые и новые ошибки. Однако это не означает, что простым разработчикам теперь не нужно искать ошибки в своих программах. Поскольку то, что дозволено Microsoft, не дозволено больше никому.

Современные приложения представляют собой достаточно сложный продукт, эксплуатация которого, как правило, производится в таких условиях, о которых и не мог подумать разработчик, поскольку фантазия пользователя неисчерпаема. Это обстоятельство приводит к тому, что всегда может возникнуть ситуация, приводящая к ошибке. Разработчик должен обнаружить эту ошибку, предпринять действия для устранения вызванных ею негативных последствий и, если это необходимо, сообщить о ней пользователю.

Изначально эта задача решалась путем анализа возвращаемых значений функций. В этом случае на каждую функцию возлагалась задача самотестирования и информирования вызывающей процедуры о возникших проблемах. Со своей стороны, вызывающая процедура должна проанализировать код ошибки, возвращаемый вызванной ею функцией и предпринять необходимые действия по обработке этой ошибки.

Такая схема может эффективно работать только в небольших приложениях, полностью создаваемых одним разработчиком, поскольку требует подробного знания всех возможных ошибок и последствий, к которым они могут привести на каждом уровне иерархии вызова. Кроме того, так может обрабатываться только ограниченный набор ошибок, поскольку расширение этого набора неизбежно приведет к замедлению выполнения программы и к снижению ее читабельности.

Возникновение объектно-ориентированного программирования придало этой проблеме новый аспект. Дело в том, что широкое распространение классов привело к использованию сложных иерархических структур памяти, управление которой осуществляется из различных объектов, что затрудняет корректное завершение программы из ее произвольной точки.

Для решения всех изложенных выше проблем используется механизм обработки исключений. Согласно этому механизму при обнаружении ошибки вызывается

исключение, которое должно быть обработано. Исключение действует как оператор возврата из функции, но этот оператор действует не только в той функции, где было вызвано исключение, но и во всех функциях, куда будет передано управление, пока не будет обнаружен обработчик данного исключения.

Таким образом, появляется возможность собрать функции обработки ошибок, возникших в каком-либо фрагменте программы, в одном месте и произвести все необходимые действия для корректного ее завершения. Учитывая, что процесс завершения работы программы может последовательно производиться в различных объектах иерархической структуры, в обработчике приложения предусмотрена возможность после завершения своей части работы, посылки нового исключения, которое будет обработано на следующем уровне иерархии.

Обработка исключений глубоко интегрирована в CLR и, если в C++ еще можно было обойтись без ее использования, то в C# это практически невозможно. В отличие от C++, в котором выделение и освобождение динамической памяти возложено на разработчика, в C# для этого используется сборщик мусора. Поэтому при обработке исключений с разработчика снимается задача освобождения памяти, выделенной уничтожаемому объекту. Это не только сокращает размер операторов обработки исключений, но и снижает вероятность возникновения ошибок при их выполнении.

Исключения предназначены для обработки особых ситуаций, поэтому при нормальной работе приложения они вызываться не должны. С другой стороны, для обеспечения единого подхода к обработке этих ситуаций в C# настоятельно не рекомендуется возвращать коды ошибок в аргументах метода.

Прежде чем перейти к описанию обработки исключений, рассмотрим наиболее часто встречающийся тип исключения — переполнение при работе с целыми числами.

Проверка переполнения

Целые числа имеют ограниченный диапазон значений, который может быть превышен при выполнении над ними арифметических операций. Такое превышение допустимого диапазона значений называется *переполнением* (*overflow*). В различных языках программирования эта ситуация обрабатывается по-разному, вплоть до полного ее игнорирования. В C# предусмотрены две возможности реагирования на возникновение переполнения. Если операция, в которой может возникнуть переполнение, помещена в оператор или в операцию `checked`, при его возникновении вызывается исключение `System.OverflowException`. Если же операция, в которой может возникнуть переполнение, помещена в оператор или в операцию `unchecked`, то исключение не вызывается, а результатом операции являются младшие биты результата, которые могут быть представлены в формате результата.

Рассмотрим обработку переполнения на следующем примере:

```
using  System;

class Demo
{
        public static void Main()
        {
                int    i = 1024;
                int    j = 4097;

                Console.WriteLine("Result is {0}", (short) (i*j) );
        }
}
```

В результате выполнения этой программы на экран будет выведена строка

```
Result is 1024
```

Отсюда можно сделать вывод, что по умолчанию целочисленные операции выполняются в контексте `unchecked`.

Изменим программу следующим образом:

```
using  System;

class Demo
{
        public static void Main()
        {
                int    i = 1024;
                int    j = 4097;

                Console.WriteLine("Result is {0}", checked((short) (i*j))
);
        }
}
```

При запуске этой программы на исполнение появится окно сообщения о возникновении в программе необработанного исключения, изображенное на рис. 15.1.

Рис. 15.1. Окно сообщения о необработанном исключении

Операторы try и catch

Теперь, когда у нас есть необработанное исключение, можно заняться его обработкой. Для этого перепишем программу следующим образом:

```
using  System;

class Demo
{
       public static void Main()
       {
         int  i = 1024;
         int  j = 4097;

         try
         {
           Console.WriteLine("Result is {0}", checked((short) (i*j)) );
              }

              catch(OverflowException e)
              {
                     Console.WriteLine( e.Message );
              }
         }
}
```

В результате выполнения этой программы на экран будет выведена строка

```
An exception of type System.OverflowException was thrown
```

Исключения могут быть вызваны одним из двух способов:

1. Для немедленного и безусловного вызова исключения используется оператор `throw`. Этот оператор относится к операторам перехода, поэтому его конечная точка недостижима.
2. Исключения могут вызываться и при выполнении некоторых операторов или выражений C#, если выполнение этих операторов или вычисление значения выражений не может быть нормально завершено. В данном случае вызывается исключение `System.OverflowException`.

При обработке и вызове новых исключений необходимо задать тип обрабатываемого или вызываемого исключения. Это может быть пользовательский тип исключения или один из системных классов исключений, перечисленных в табл. 15.1.

Таблица 15.1. *Системные классы исключений*

Класс исключения	Описание
`System.ArgumentException`	Базовый класс исключений, связанных с аргументами
`System.ArgumentNullException`	Вызывается методами, которым передали аргумент null, в том случае, если это недопустимо
`System.ArgumentOutOfRangeException`	Вызывается методами, аргумент которых вышел из заданного диапазона значений
`System.ArithmeticException`	Базовый класс исключений, возникающих в процессе выполнения арифметических операций. Таких, как DivideByZeroException и OverflowException
`System.ArrayTypeMismatchException`	Вызывается при попытке сохранения в массиве значения, тип которого несовместим с типом элемента массива
`System.COMException`	Исключения, содержащие коды завершения объектов COM в формате HRESULT
`System.DivideByZeroException`	Вызывается при попытке деления целочисленной величины на ноль
`System.Exception`	Базовый класс для всех объектов исключений
`System.IndexOutOfRangeException`	Вызывается при обращении к элементу массива по индексу, значение которого меньше нуля или превышает верхнюю границу массива
`System.IteropException`	Базовый класс исключений, вызываемых за пределами среды CLR
`System.InvalidCastException`	Вызывается при невозможности выполнения явного преобразования из базового типа или типа интерфейса в производный тип в процессе выполнения приложения
`System.InvalidOperationException`	Вызывается методами в том случае, если их вызов недопустим в данном контексте

Таблица 15.1 *(окончание)*

Класс исключения	Описание
`System.MulticastNotSupportedException`	Вызывается при попытке объединить два ненулевых делегата, тип возвращаемого значения которых отличен от типа void
`System.NullReferenceException`	Вызывается при попытке получения доступа к объекту по ссылке, имеющей значение null
`System.OutOfMemoryException`	Вызывается при невозможности выделения памяти операцией new
`System.OverflowException`	Вызывается в арифметических операциях при использовании оператора или операции checked при возникновении переполнения
`System.SEHException`	Исключения, содержащие информацию, передаваемую при структурной обработке исключений Win32
`System.StackOverflowException`	Вызывается при переполнении стека большим числом взаимосвязанных вызовов методов. Обычно свидетельствует о глубокой или бесконечной рекурсии
`System.SystemException`	Базовый класс для всех объектов исключений, генерируемых в процессе выполнения приложения
`System.TypeInitializationException`	Вызывается в статическом конструкторе для необработанных типов исключений

Программный текст, в котором может быть вызвано исключение, помещается в оператор `try`, непосредственно за которым должен размещаться оператор `catch`, в котором и производится обработка данного исключения. Аргумент оператора `catch` имеет тип объекта обрабатываемого исключения, для которого могут быть вызваны его методы, и через который можно получить доступ к его полям и свойствам.

В приведенном примере обрабатывается исключение `System.OverflowException`, являющееся потомком класса `System.Exception`. Этот класс содержит, помимо прочего, два доступных только для чтения свойства, к которым можно обратиться из любого типа исключения. Значения этих свойств могут быть заданы в конструкторе класса исключения.

- Свойство `Message` содержит текстуальное описание причины вызова исключения.

- Свойство `InnerException` содержит "внутреннее исключение" для данного исключения. Если это поле не имеет значения `null`, то это исключение было вызвано при обработке другого исключения. Доступ к нему обеспечивается свойством `InnerException`.

Обработка исключений производится в операторе `try`. При вызове исключения в этом блоке система производит поиск ближайшего блока `catch`, обрабатывающего данный тип исключений. Сначала поиск производится в том операторе `try`, в котором было вызвано исключение. При этом все обработчики `catch` просматриваются в порядке их расположения в операторе `try`. Если этот поиск завершился безрезультатно, поиск производится в операторе `try`, включающем в себя текущий оператор `try`. Если метод не содержит внешних операторов `try`, поиск производится в методе, его вызвавшем. При этом полагается, что вложенный оператор `try` располагается в месте вызова метода. Считается, что обработчик `catch` обрабатывает данный тип исключения, если тип обрабатываемого им исключения совпадает с типом вызванного исключения или является его базовым типом. Если в обработчике `catch` не указан тип исключения, то он обрабатывает исключение типа `System.Exception`, то есть исключения любого типа.

Если в результате выполнения описанной выше процедуры был найден обработчик исключения, то управление передается данному обработчику. Если в операторе `try` не было обнаружено нужного обработчика `catch`, то перед тем, как передать управление внешнему оператору `try`.

Если обработчик вызванного типа исключения не был найден, то:

- если поиск привел в статический конструктор или инициализатор статического поля, в достигнутой точке статического конструктора или инициализатора вызывается исключение `System.TypeInitializationException`. Свойство `InnerException` данного исключения возвращает исходное вызванное исключение;
- если поиск привел к программному коду, инициирующему поток или программу, выполнение данного потока или программы завершается.

Такая процедура обработки исключения связана с тем, что в большинстве случаев в методе, в котором было вызвано исключение, нельзя произвести его полную обработку, поскольку в нем отсутствует вся необходимая для этого информация. При передаче исключения на каждом уровне иерархии объектов с ним могут быть произведены следующие действия:

- оно может быть транслировано на следующий уровень иерархии;
- оно может быть обработано;
- может быть создано новое исключение.

Трансляция исключений

Эта операция применяется, если для завершения работы объекта, которому передано данное исключение, не требуется производить никаких дополнительных действий. В этом случае в нем отсутствует оператор `try`, включающий в себя программный код, в котором было вызвано исключение и, соответственно, связанные с ним операторы `catch`.

Обработка исключения

Если для корректного завершения работы объекта требуется произвести некоторые действия, то код, в котором может быть вызвано исключение, помещается в оператор `try`, после которого располагаются операторы `catch`, обрабатывающие возникшие исключения. Если эта обработка не является окончательной, и существует необходимость обработки исключения на следующих уровнях иерархии объектов, в блоке оператора `catch` вызывается исключение того же типа. К обработке исключения в объекте предъявляется требование, чтобы после ее завершения объект оставался бы в допустимом состоянии. Это связано с тем, что в C# отсутствует возможность уничтожения объекта, и он будет существовать до тех пор, пока его не уничтожит сборщик мусора.

Рассмотрим этот вариант обработки исключения на примере:

```
using  System;

class MyClass
{
        public MyClass(int i, int j)
        {
                this.i = i;
                this.j = j;
        }

        public int     Divide()
        {
                checked
                {
                        try
                        {
                                return (i/j);
                        }

                        catch(DivideByZeroException e)
                        {
```

```
                                Console.WriteLine("Internal processing");
                                throw  e;
                        }
                }
        }

        int     i;
        int     j;
}

class Demo
{
        public static void Main()
        {
                MyClassmc       = new MyClass( 1, 0 );

                try
                {
                        int     i = mc.Divide();
                }

                catch(Exception e)
                {
                        Console.WriteLine( e.Message );
                }
        }
}
```

В результате выполнения этой программы на экран будет выведен следующий текст:

```
Internal processing
Attempted to divide by zero.
```

Создание нового исключения

В некоторых случаях возникает необходимость изменить тип посылаемого исключения (например, на пользовательский) или передать в нем сообщение, отличное от того, которое передается по умолчанию.

Пример посылки такого исключения приведен ниже:

```
using  System;

class MyClass
{
```

```
        public MyClass(int i, int j)
        {
                this.i = i;
                this.j = j;
        }

        public int    Divide()
        {
                checked
                {
                        try
                        {
                                return (i/j);
                        }

                        catch(DivideByZeroException e)
                        {
                          Console.WriteLine("Internal processing");
                          throw(new DivideByZeroException("Divider is
                         null", e));
                        }
                }
        }

        int    i;
        int    j;
}

class Demo
{
        public static void Main()
        {
                MyClassmc     = new MyClass( 1, 0 );

                try
                {
                        int    i = mc.Divide();
                }

                catch(Exception e)
                {
```

```
                        Console.WriteLine( e.Message );
                }
        }
}
```

В результате выполнения этой программы на экран будет выведен следующий текст:

```
Internal processing
Divider is null
```

Пользовательские классы исключений

В приведенном выше примере создания нового исключения типы нового и исходного исключения совпадают. В большинстве случаев разработчику требуется не только модифицировать значения полей существующего класса исключения, но и передать свою информацию, необходимую для более качественной обработки исключения. Эта задача может быть решена при создании пользовательского класса исключения.

Ниже приведен пример использования такого класса:

```
using  System;

class CustomException: Exception
{
        public CustomException()
        {
        }
        public CustomException(string message): base(message)
        {
        }
        public CustomException(string message, Exception inner):
base(message, inner)
        {
        }
}

class MyClass
{
        public MyClass(int i, int j)
        {
                this.i = i;
```

```
            this.j = j;
        }

        public int    Divide()
        {
            checked
            {
                try
                {
                    return (i/j);
                }

                catch(DivideByZeroException e)
                {
                    Console.WriteLine("Internal processing");
                    throw  (new CustomException("Divider is
                     null", e));
                }
            }
        }

        int    i;
        int    j;
}

class Demo
{
        public static void Main()
        {
            MyClassmc     = new MyClass( 1, 0 );

            try
            {
                int    i = mc.Divide();
            }

            catch(Exception e)
            {
                Console.WriteLine( e.Message );
            }
        }
}
```

Эта программа выводит на экран те же сообщения, что и ее предыдущая версия, но в данном случае разработчик может надежно идентифицировать источник возникновения исключения и предпринять соответствующие действия.

Оператор finally

В некоторых объектах существуют действия, которые необходимо предпринять до завершения их работы независимо от того, произошло ли это вследствие естественных причин или вследствие вызова исключения. К таким действиям относится, например, закрытие файла. Такие операции можно было бы поместить в деструктор объекта, однако использование деструкторов в C# нежелательно и, кроме того, может возникнуть ситуация, когда эти операции должны быть непременно выполнены до того, как управление будет передано на следующий уровень иерархии объектов.

Конечно, можно было бы поместить этот код в каждый из операторов `catch`, а также в конец метода, но такой подход, во-первых, увеличивает размер программы, во-вторых, снижает ее читабельность и, в-третьих, не гарантирует того, что во всех ветвях исполнения программы будут выполняться одни и те же операции завершения работы объекта.

Для решения этой проблемы в C# используется оператор `finally`, располагающийся после всех операторов `catch`, если они присутствуют, или после оператора `try`, если операторы `catch` отсутствуют.

Пример использования оператора `finally` приведен ниже:

```
using  System;
using  System.IO;

class MyClass
{
       public MyClass(int i, int j)
       {
              this.i = i;
              this.j = j;
       }

       public void   Divide()
       {
              checked
              {
                     FileStream file = new FileStream("Data.txt",
                                                      FileMode.Create );

                     try
                     {
```

```
                        StreamWriter sw = new StreamWriter( file );

                        sw.WriteLine( i/j );
                    }

                    catch(DivideByZeroException e)
                    {
                        Console.WriteLine("Internal processing");
                        throw  e;
                    }

                    finally
                    {
                        file.Close();
                    }
                }
            }

            int    i;
            int    j;
}

class Demo
{
            public static void Main()
            {
                MyClassmc     = new MyClass( 1, 0 );

                try
                {
                    mc.Divide();
                }

                catch(Exception e)
                {
                    Console.WriteLine( e.Message );
                }

                Console.ReadLine();
            }
}
```

В данном примере метод `MyClass.Divide` не возвращает результат деления, а сохраняет его в файле. Поэтому, как после обработки исключения, так и после нормального завершения работы метода, в нем необходимо закрыть файл, в котором сохранялись результаты.

Обратите внимание на то, что объект класса `FileStream` создается вне блока оператора `try`, хотя подобные операции являются одним из основных источников вызова исключений. Это связано с тем, что при создании этого объекта в операторе `try` он окажется за пределами области видимости оператора `finally`, что немедленно приведет к сообщению об ошибке.

Глава 16

Обеспечение безопасности

Как уже говорилось выше, одной из целей создания C# является обеспечение безопасности приложений. Пока что этот вопрос затрагивался только в связи с безопасностью типов, однако, в C# предусмотрены и специальные меры по обеспечению безопасности распространяемых в сети приложений.

Поскольку Internet является открытой системой, не все пользователи которой отличаются кристальной честностью, компоненты, поступающие из данного источника, должны подвергаться тщательной проверке на предмет безопасности их использования. В связи с этим в CLR средства безопасности существуют не только на уровне кода, но и на уровне прав доступа пользователей.

Безопасность доступа к компоненту

В настоящее время программный код может поступать к пользователю не только с сервера компании, где, по идее, должен храниться только абсолютно надежный код, но и из таких сомнительных источников, как Internet, электронная почта и CD (которые "случайно" могут оказаться пиратскими). Причем, как легко можно догадаться, разработчиков программного обеспечения куда больше беспокоят упущенные ими прибыли, чем безопасность работы пользователя, но у пользователей, являющихся потенциальными покупателями их продукции, противоположный взгляд на эту ситуацию. Поэтому проблемы безопасности одновременно развиваются в обоих этих направлениях.

При обеспечении безопасности доступа к коду контролируется доступ к защищенным ресурсам и операциям. При этом проверяется его источник и наличие соответствующих сертификатов.

Ниже приведены основные принципы обеспечения безопасности доступа к коду:

- сетевые администраторы могут задавать различные права доступа для различных компонент;
- вызываемый компонент может проверять вызывающий процесс на наличие у него прав доступа к нему;

- компонент может исполняться только в среде, обеспечивающей достаточный уровень безопасности его выполнения;
- компонент может запрашивать права, требуемые для его исполнения, а также в явном виде указывать права, которыми он не должен обладать;
- доступ к различным системным ресурсам контролируется правами доступа, предоставленными компоненту;
- право доступа может быть предоставлено компоненту во время его загрузки по специальному запросу или на основе разрешений политики безопасности.

Одним из следствий приведенного набора правил является то, что программа, обладающая меньшими правами доступа, не может вызвать компонент, обладающий большими правами доступа. Этот принцип позволяет относительно безопасно использовать компоненты, загружаемые через Internet.

Проверка безопасности компонента сводится к двум основным проверкам:

- проверке безопасности типов компонента;
- проверке запрашиваемых им прав доступа.

Безопасность типов

Требование безопасности типов является минимальным требованием, предъявляемым к коду, который претендует на звание безопасного. Безопасность типов включает в себя, помимо прочего, проверку прав определенных в этом типе функциональных членов. Эти члены типов не должны содержать "дыр безопасности", позволяющих вызывать их из программ, имеющих меньшие права доступа, чем предоставлены данному типу. Для обеспечения этого требования достаточно, чтобы любой доступ к типам осуществлялся бы только проверенными методами, а сами эти методы отвечали бы требованиям безопасности.

Весь создаваемый в C# код, кроме явным образом помеченного как небезопасный (`unsafe`), является безопасным. Поэтому прежде чем заверить безопасность кода, в среде CLR производится проверка его IL-кода и метаданных.

Права доступа

После проверки безопасности типов среда CLR переходит к проверке *прав доступа* (permissions). Эта проверка позволяет определить, что данный код может получить все права доступа, которые необходимы для его выполнения. В некоторых случаях код может запросить права, без которых он может исполняться, но наличие которых позволяет существенно повысить его эффективность. Использование компонентом только необходимых ему прав доступа гарантирует, что он может исполняться в среде со строгим ограничением прав доступа.

Все права доступа разделяются на три группы:

- *необходимые* (required) — права, без которых компонент не может работать;
- *необязательные* (optional) — права, не являющиеся обязательными, но наличие которых существенно повышает эффективность приложения;
- *запрещенные* (refused) — права, которые компонент не должен иметь даже в том случае, если это допускается политикой безопасности. Задание этого набора прав позволяет снизить уязвимость компонента.

Некоторые из этих прав запрашиваются кодом или присваиваются ему, а другие он получает от пользователя.

Стандартные права доступа

Стандартные права доступа (standard permissions) используются для грубой группировки различных компонентов по их правам доступа к защищенным ресурсам и операциям. Список этих прав и краткое их описание приведено в табл. 16.1.

Таблица 16.1. *Стандартные права доступа*

Право доступа	Описание
EnvironmentPermission	Определяет права доступа к переменным окружения. Позволяет задать один из двух возможных типов доступа: только для чтения (read only) или для записи (write). Право доступа для записи включает в себя право создавать и уничтожать переменные окружения
FileDialogPermission	Вызывает системное диалоговое окно, в котором пользователь должен разрешить доступ к файлу
FileOPermission	Позволяет задать режим доступа к файлу: чтение (read), запись (write) и добавление в конец (append)
FidoPermission	При задании этого режима право доступа на чтение расширяется возможностью обращения к информации о файле, право доступа на запись расширяется возможностью удаления и перезаписи файла, а право на запись в конец файла не позволяет выполнять чтение
IsolatedStoragePermission	Управляет доступом данного пользователя к изолированному (isolated) дисковому пространству. Позволяет ограничить режим использования и размер выделяемого дискового пространства, а также задавать время хранения или момент истечения срока хранения данных
ReflectionPermission	Задает режим доступа по чтению к информации о типах закрытых членов типов
RegistryPermission	Задает режим чтения, записи и создания новых ключей в системный реестр. Режим доступа к ключам и записям в системном реестре должны задаваться раздельно

Таблица 16.1 *(окончание)*

Право доступа	Описание
SecurityPermission	Содержит набор флагов доступа, используемых системой безопасности. Эти ключи позволяют задавать режимы выполнения кода, перегрузки и отмены проверок в системах безопасности, вызова небезопасного кода, определять последовательность выполнения и решать другие задачи
UIPermission	Определяет режим доступа к различным элементам пользовательского интерфейса, включая окна, события и буфер обмена

Персональные права доступа

Персональные права доступа (identity permissions) не запрашиваются приложением, а предоставляются на основании сертификата, сопровождающего данный компонент. Эти права позволяют определить происхождение и автора контролируемого приложения на основании включенной в сертификат цифровой подписи.

Персональные права доступа и их описание приведены в табл. 16.2.

Таблица 16.2. *Персональные права доступа*

Право доступа	Описание
PublisherIdentityPermission	Подпись компоненты .NET подтверждает издателя
StrongNameIdentityPermission	Определяет криптостойкое имя компоненты. Идентификатор состоит из криптостойкого и простого имен
ZoneIdentityPermission	Определяет зону происхождения компонента. Зона может не совпадать с зоной, определяемой URL
SiteIdentityPermission	Определяет права, предоставляемые Web-страницей, с которой был загружен компонент
URLIdentityPermission	Определяет права, предоставляемые URL, с которого был загружен компонент

Ролевая безопасность

Система ролевой безопасности .NET в какой-то степени похожа на соответствующую систему, используемую в COM, однако, между этими системами существуют и некоторые отличия.

Ролевая безопасность .NET основана на модели *исполнителя* (principal), действующего от имени самого пользователя или другого исполнителя того же

самого пользователя. Среда CLR принимает решение по каждому исполнителю на основании его идентификатора и ролей, в которых он участвует.

Под *ролью* (role) понимается именованный набор пользователей, обладающих одинаковыми привилегиями. Один и тот же исполнитель может одновременно играть несколько ролей, определяющих допустимость выполнения запрошенных им действий.

Различают три типа исполнителей:

- *обычные пользователи* (generic principals) — к ним относятся неавторизованные пользователи, а также доступные им роли;
- *исполнители Windows* (Windows principals) — к ним относятся пользователи и роли Windows. Для них поддерживается режим доступа к ресурсу от имени другого пользователя (имитация);
- *специальные исполнители* (custom principals) — эти исполнители определяются в приложении. Они могут расширять понятия идентификатора исполнителя и выполняемых им ролей. Приложение, использующее данный тип исполнителей, должно содержать модуль идентификации.

Для проверки прав доступа исполнителей в C# используется класс `PrincipalPermission`. Он позволяет среде исполнения осуществлять проверку прав исполнителя, аналогичную той, которая производится для доступа к коду. Кроме того, он позволяет приложению получать всю необходимую информацию для организации собственной проверки.

Приложение

Интерфейс пользователя Visual Studio.NET

Разработка приложений C# производится в среде Visual Studio.NET, все компоненты которой используют единый интерфейс пользователя. Интерфейс Visual Studio представляет собой существенно измененный интерфейс Developer Studio версий 5.0 и 6.0.

Окно Visual Studio состоит из нескольких независимых окон, обычно отображаемых в виде панелей, и панелей инструментов, являющихся, по сути, полноценными окнами. Список панелей инструментов можно получить, щелкнув правой кнопкой мыши по любой отображаемой панели инструментов. Этот список представляет собой контекстное меню, позволяющее выбирать окна и панели инструментов, отображаемые в данный момент на экране. Окно Visual Studio со стандартным набором окон и раскрытым контекстным меню показано на рис. П.1.

Чтобы стандартные окна и панели инструментов перестали отображаться в виде панелей, а превратились в полноценные окна, их необходимо переместить в центр окна Visual Studio. Однако в этом практически никогда не возникает необходимости. Стандартные окна Visual Studio.NET объединены в панели. Чтобы превратить эту панель в полноценное окно достаточно дважды щелкнуть левой кнопкой мыши на ярлыке этой панели.

Часто этот двойной щелчок происходит непреднамеренно, и разработчику приходится превращать это окно обратно в панель, что представляет собой достаточно неприятную задачу, особенно при отсутствии опыта. В большинстве случаев, для того чтобы превратить самостоятельное окно в панель достаточно поместить указатель мыши на заголовок этого окна и переместить его на заголовок окна, вкладкой которого должно стать текущее окно. При этом рамка перетаскиваемого окна принимает форму вкладки. После вставки новой вкладки в окно порядок следования ярлыков вкладок можно изменить простым перетаскиванием.

Иногда разработчик случайно уничтожает окно с вкладками (для этого достаточно нажать кнопку **Close** окна) или объединяет два окна с вкладками. Для восстановления окна с вкладками лучше всего сначала собрать это окно (содержащее хотя бы две вкладки) в самостоятельном окне (используя описанную выше методику), а затем перетащить его в новую позицию, где оно

превратится в панель окна Microsoft Visual C++ (среда разработки почему-то считает приложение C# приложением C++) .

Рис. П.1. Окно Visual Studio.Net

Интерфейс пользователя Visual Studio.NET включает в себя комплекс, состоящий из окон, инструментальных средств, меню, панелей инструментов и других элементов, позволяющих использовать эту среду программирования для создания и отладки приложения. Кроме того, Visual Studio.NET позволяет работать и с документами других приложений таких, как Microsoft Excel или Microsoft Word. Для этого используется технология ActiveX. Поскольку все перечисленные приложения созданы одной фирмой, которая к тому же разработала технологию ActiveX, то можно надеяться, что работа с этими документами не подвесит систему.

Visual Studio.NET позволяет настроить свой интерфейс в соответствии с предпочтениями пользователя, который может создать собственные панели инструментов, меню и комбинации клавиш. Однако этим не стоит злоупотреблять, поскольку, привыкнув к своей нестандартной конфигурации, вы будете испытывать серьезные затруднения при работе в стандартной конфигурации панелей, установленной на других машинах, что может существенно повредить вашей репутации программиста.

Первая страница Visual Studio.NET

При первом запуске Visual Studio.NET пользователь выходит в первую страницу этого приложения, изображенную на рис. П.2 (на этом рисунке приведен не исходный вид страницы, а ее вид после нескольких сеансов работы с приложением).

Рис. П.2. Первая страница Visual Studio.NET

На раскрывающейся при вызове этой страницы вкладке **Get Started** (Начало работы) имеется возможность открыть текущий, существующий или создать новый проект, а также просмотреть отчет об ошибках Visual Studio.NET. На этой же странице расположен флажок **Show the start page on next startup** (Показывать первую страницу при следующем запуске). Как следует из его названия, если этот флажок не сброшен, первая страница будет появляться при каждом запуске Visual Studio.NET. Поскольку в большинстве случаев можно обойтись и без предоставляемых данной страницей услуг, этот флажок лучше всего сразу же сбросить. При необходимости получить доступ к первой странице в процессе работы с Visual Studio.NET, достаточно выбрать команду меню **Help**, **Show Start Page** (Справка, Вывести первую страницу).

Вкладка **What's New** первой страницы позволяет получить информацию о новых свойствах данной среды разработки, вкладка **Online Community** позволяет получить информацию по различным вопросам разработки приложений, размещенную в Internet, вкладка **Headlines** предоставляет доступ к интерактивной справочной системе MSDN, также размещенной в Internet, а вкладка **Search Online** позволяет производить поиск в Сети.

Особое внимание следует обратить на вкладку **My Profile**, приведенную на рис. П.3. Эта вкладка позволяет настроить среду программирования Visual Studio.NET в соответствии с требованиями конкретного разработчика.

Рис. П.3. Настройка профиля разработчика

Эта вкладка состоит из нескольких раскрывающихся списков, имеющих следующее назначение:

- **Profile:** — позволяет выбрать профиль в соответствии с используемым для разработки языком программирования.
- **Keyboard Scheme:** — позволяет выбрать разработчику привычную раскладку клавиатуры.
- **Windows Layout:** — определяет расположение окон в среде разработки.

- **Help Filter:** — ограничивает объем выводимой справочной информации.
- **At Startup Show:** — определяет, что именно будет выводиться на экран при запуске среды программирования.

Кроме того, вкладка содержит переключатель **Show Help:**, определяющий способ вывода справочной информации в среде программирования. По умолчанию этот переключатель установлен в положение **Integrated**. При таком положении переключателя справочная информация выводится в стиле Visual C++ 5.0, то есть в окно среды программирования. При установке данного переключателя в положение **Outside the IDE** справочная информация выводится в стиле Visual C++ 6.0, то есть в отдельное приложение. Если переключение в положение **Outside the IDE** было произведено после вывода информации в окно среды программирования, то соответствующие окна (например, окно **Index**) в среде программирования не убираются автоматически, что приводит к конфликтам при попытке их использования.

Панели инструментов

При первом запуске среды программирования Visual Studio.NET, использующей стандартную конфигурацию панелей инструментов, непосредственно под панелью меню отображается стандартная панель инструментов. Она содержит кнопки, необходимые при работе практически в любом режиме функционирования Visual Studio.NET. При работе пользователя в редакторе ресурсов, как правило, на экран выводятся дополнительные панели инструментов.

Конкретный набор панелей инструментов, отображаемых на экране, выбирается пользователем, исходя из его предпочтений и используемого им разрешения дисплея. Помимо выбора отображаемых панелей инструментов пользователь может выбрать и содержимое каждой панели инструментов. Ему предоставлена возможность убрать любую кнопку из стандартной панели инструментов и добавить в нее кнопки из некоторого набора.

Чтобы выбрать набор панелей инструментов, отображаемых на экране:

1. Щелкните правой кнопкой мыши по любой выведенной на экран панели инструментов.
2. В появившемся контекстном меню установите или сбросьте флажок у соответствующей панели инструментов. На экране появится или с экрана исчезнет данная панель инструментов.
3. Повторите п.п. 1 и 2 для других панелей инструментов.

Или:

1. Выберите команду меню **Tools**, **Customize** (Сервис, Настройка). Появится диалоговое окно **Customize** (Настройка).
2. Раскройте вкладку **Toolbars** (Панели инструментов). Диалоговое окно **Customize** примет вид, изображенный на рис. П.4.

Рис. П.4. Диалоговое окно **Customize**, вкладка **Toolbars**

3. Установите или сбросьте флажки в окне списка **Toolbars:**. При установке или сбросе каждого флажка в окне будет появляться или исчезать соответствующая панель инструментов.
4. После установки нужной конфигурации нажмите кнопку **Close**.

Чтобы добавить или удалить кнопку в панели инструментов:

1. Установите требуемый набор панелей инструментов, как это описано выше.
2. Выберите команду меню **Tools**, **Customize**. Появится диалоговое окно **Customize**.
3. Раскройте вкладку **Commands** (Команды). Диалоговое окно **Customize** примет вид, изображенный на рис. П.5.
4. Поместите указатель мыши на кнопку панели инструментов, которую необходимо удалить.
5. Нажмите левую кнопку мыши и перетащите эту кнопку за пределы панели инструментов.
6. Отпустите левую кнопку мыши. Данная кнопка будет удалена из панели инструментов.
7. Поместите указатель мыши на кнопку панели инструментов, которую необходимо переместить в другую панель инструментов.
8. Нажмите левую кнопку мыши и перетащите эту кнопку в другую панель инструментов.
9. Отпустите левую кнопку мыши. Данная кнопка переместится в другую панель инструментов.

Рис. П.5. Диалоговое окно **Customize**, вкладка **Commands**

10. В раскрывающемся списке **Categories:** (Категории) выберите категорию, содержащую новую кнопку, которую необходимо поместить в панель инструментов. (Категория обычно соответствует меню, включающему в себя необходимую команду или панели инструментов, уже содержащие данную кнопку).
11. Нужная кнопка появится в окне списка **Commands:** (Команды).
12. Поместите указатель мыши на значок новой кнопки в окне списка **Commands:**.
13. Нажмите левую кнопку мыши и перетащите эту кнопку в ту панель инструментов и в ту позицию в панели инструментов, где она должна находиться.
14. Отпустите левую кнопку мыши. Данная кнопка появится в панели инструментов.
15. После установки нужной конфигурации нажмите кнопку **Close**.

Любая из панелей инструментов может быть сделана плавающей или зафиксирована на любой из границ рабочей области, как это показано на рис. П.6.

Для перемещения зафиксированной панели инструментов необходимо поместить указатель мыши на ее *бордюр* (wrinkles), представляющий собой едва заметную штриховую полоску на одном из ее краев, и, нажав левую кнопку мыши, переместить ее в новое положение. Если панель инструментов находится далеко от границ рабочей области, она становится плавающей, то есть превращается в обычное окно, как это показано на рис. П.6 для панели ин-

струментов **Text Editor**. При приближении перемещаемой панели к границе рабочей области изменение формы ее рамки указывает на то, что она будет зафиксирована.

Рис. П.6. Размещение панелей инструментов

При установке зафиксированных панелей для устранения промежутков между ними рекомендуется "наезжать" устанавливаемой панелью инструментов на уже установленную. При этом Visual Studio сама позаботится об устранении перекрытия панелей и установит их встык.

Наиболее часто пользователю придется работать с панелью инструментов **Standard**. Ниже приведено ее краткое описание.

Панель инструментов Standard

Стандартная панель инструментов позволяет пользователю осуществлять вызов, редактирование и сохранение файлов. Кнопки данной панели инструментов, как правило, дублируются командами меню. В табл. П.1 приведено соответствие кнопок панели инструментов **Standard** командам меню.

Таблица П.1. *Соответствие кнопок панели инструментов Standard командам меню*

Кнопка панели инструментов	Команда меню
New Project	**File, New, Project...**
Add New Item	**Project, Add New Item...**
Open File	**File, Open**
Save	**File, Save**
Save All	**File, Save All**
Cut	**Edit, Cut**
Copy	**Edit, Copy**
Paste	**Edit, Paste**
Undo	**Edit, Undo**
Redo	**Edit, Redo**
Navigate Backward	**View, Navigate Backward**
Navigate Forward	**View, Navigate Forward**
Start	**Debug, Start**
Find in Files	**Edit, Find and Replace, Find in Files**
Solution Explorer	**View, Solution Explorer**
Properties Window	**View, Properties Window**
Toolbox	**View, Toolbox**
Class View	**View, Class View**

Следует обратить внимание на то, что кнопки **New Project**, **Add New Item** и **Class View** представляют собой раскрывающиеся меню, первая команда которых соответствует их названию и сопоставленной им команде основного меню, а остальные команды позволяют производить некоторые дополнительные операции.

Кнопка **Navigate Backward** также представляет собой раскрывающееся меню, в котором перечислены последние операции перемещения курсора между реализациями различных типов, что позволяет непосредственно выбрать отменяемую операцию. Если текущее положение курсора было установлено при переключении из другого окна, то при нажатии кнопки **Navigate Backward** будет осуществлен переход в это окно.

Помимо кнопок панель инструментов **Standard** содержит два раскрывающихся списка. Раскрывающийся список **Solution Configuration** позволяет ус-

тановить конфигурацию создаваемого приложения (отладочную или распространяемую), а раскрывающийся список **Find** позволяет задать искомый фрагмент текста при использовании команды меню **Edit, Find and Replace, Find**.

Система меню

Среда разработки Visual Studio.NET имеет достаточно сложную систему меню. Некоторые команды находятся на третьем или даже четвертом уровне иерархии. Обычно этим командам соответствует некоторая комбинация клавиш, позволяющая вызвать их значительно быстрее, чем с использованием меню. У пользователя Visual Studio.NET есть более важные дела, чем запоминать несколько десятков комбинаций клавиш, большую часть из которых он будет использовать не чаще, чем раз в месяц.

Однако разработчики Visual Studio.NET были убеждены в обратном и установили такое количество комбинаций клавиш, что практически любое ошибочное нажатие вызывает какое-нибудь диалоговое окно, что не дает пользователям данной среды программирования забыть хорошую русскую пословицу: "Заставь дурака богу молиться, он и лоб расшибет".

Панель меню Visual Studio.NET содержит девять заголовков меню, которые перечислены ниже:

- **File** — меню операций, связанных с файлами, включая их создание, открытие, закрытие и печать.
- **Edit** — меню операций редактирования, включая копирование, удаление, восстановление измененной информации, поиск и перемещение по файлу.
- **View** — меню управления способом отображения информации на экране, включая управление отображением панелей инструментов и окон среды разработки **Output** и **Workspace**.
- **Project** — меню команд управления разрабатываемым проектом в целом.
- **Build** — меню команд компиляции и компоновки приложения.
- **Debug** — меню команд отладки приложения и запуска его на исполнение.
- **Tools** — меню команд настройки Visual C++ и доступа к автономным утилитам.
- **Window** — меню выбора окон и их автоматического упорядочения на экране.
- **Help** — меню команд обращения к справочной системе Visual C++.

Ниже будут рассмотрены все перечисленные выше меню и описаны их основные команды.

Меню File

Меню **File**, приведенное на рис. П.7, содержит команды, осуществляющие операции, связанные с файлами, включая их создание, открытие, закрытие и печать.

Рис. П.7. Меню **File**

Команда File, New

Данная команда не является терминальной: при ее выборе появляется контекстное меню, содержащее две команды.

При выборе команды меню **File**, **New**, **Project...** (Файл, Создать, Проект) появляется диалоговое окно **New Project** , изображенное на рис. П.8.

Это диалоговое окно используется для создания новых проектов. Его использование подробно описано в главе 2. Вместо данной команды меню можно использовать комбинацию клавиш **<Ctrl>+<Shift>+<N>**.

См. главу 2.

При выборе команды меню **File**, **New**, **File...** (Файл, Создать, Файл) появляется диалоговое окно **New File** , изображенное на рис. П.9.

Рис. П.8. Диалоговое окно **New Project**

Рис. П.9. Диалоговое окно **New File**

Это диалоговое окно используется для создания в проекте пустых файлов программ, описаний рабочей среды или других документов. Вместо этой команды меню можно использовать комбинацию клавиш **<Ctrl>+<N>**.

Команда File, Open

Данная команда меню также не является терминальной, а при ее выборе появляется контекстное меню, содержащее четыре команды.

При выборе команды меню **File**, **Open**, **Project...** (Файл, Открыть, Проект) появляется диалоговое окно **Open Project** , изображенное на рис. П.10.

Рис. П.10. Диалоговое окно **Open Project**

Это диалоговое окно содержит переключатель, расположенный по умолчанию в положении **Close Solution**. При таком положении переключателя при открытии нового проекта старое Решение (Solution) закрывается, и проект размещается в новом решении. При нахождении переключателя в положении **Add to Solution** новый проект помещается в старое решение.

Вместо команды меню можно использовать комбинацию клавиш **<Ctrl>+<Shift>+<O>**.

При выборе команды меню **File**, **Open**, **Project From Web...** (Файл, Открыть, Проект в Сети) появляется диалоговое окно **Open Project From Web**, изображенное на рис. П.11.

Рис. П.11. Диалоговое окно **Open Project**

Это диалоговое окно содержит раскрывающийся список, в текстовом поле которого следует указать адрес сервера, где следует открыть проект.

При выборе команды меню **File**, **Open**, **File...** (Файл, Открыть, Файл) появляется диалоговое окно **Open File**, изображенное на рис. П.12.

Рис. П.12. Диалоговое окно **Open File**

Это диалоговое окно используется для включения в проект файлов программ, описаний рабочей среды или других документов и представляет собой стандартное диалоговое окно **Windows File Open**, используемое практически во всех приложениях Windows. В раскрывающемся списке **Files of type:** указываются типы файлов, отображаемых в окне списка диалогового окна. Если вам необходимо открыть файл, не включенный в список, выберите в раскрывающемся списке комбинацию, содержащую нужный вам тип файлов. При этом произойдет изменение содержимого окна списка данного диалогового окна.

Рис. П.13. Диалоговое окно **Open With**

Раскрывающийся список кнопки **Open** позволяет изменить режим открытия файла. Чтобы выбрать режим открытия файла необходимо выбрать в нем команду **Open With...** (Открыть как). Появится диалоговое окно **Open With**, изображенное на рис. П.13.

Вместо команды меню **Open With** можно использовать комбинацию клавиш **<Ctrl>+<O>**.

При выборе команды меню **File**, **Open**, **File From Web...** (Файл, Открыть, Файл в Сети) появляется диалоговое окно **Open File From Web**, аналогичное окну **Open Project From Web,** изображенному на рис. П.11.

Команда File, Close

Выбор данной команды меню приводит к закрытию файла, окно которого имеет на данный момент фокус ввода. Если таких файлов нет, то команда **File, Close** становится недоступной. Другим способом закрыть файл и связанное с ним окно можно, нажав на кнопку **Закрыть** (кнопка с косым крестиком, расположенная в крайней правой позиции заголовка окна).

Команда File, Add Project

Эта команда меню не является терминальной, а при ее выборе появляется контекстное меню, содержащее две команды. Команда **File**, **Add Project, New Project...** (Файл, Добавить проект, Новый проект) аналогична описанной выше команде **File**, **New**, **Project...**, а команда **File**, **Add Project, Existing Project...** (Файл, Добавить проект, Существующий проект) аналогична описанной выше команде **File**, **Open**, **Project...**. Однако в обоих случаях проект будет включен в текущее решение. Поэтому диалоговое окно **Add Existing Project**, в отличие от окна **Open Project**, не содержит переключателя, позволяющего закрыть текущее приложение.

Команда File, Open Solution

Выбор данной команды меню позволяет открыть существующее решение. При этом появится диалоговое окно **Open Solution**, изображенное на рис. П.14.

Рис. П.14. Диалоговое окно **Open Solution**

Чтобы открыть новое решение, раскройте папку, содержащую нужное решение, выделите файл решения и нажмите кнопку **Open**. Если текущее решение содержит несохраненную информацию, на дисплее появится соответствующее сообщение. После этого в среду программирования Visual Studio.NET будет загружено новое решение.

Команда File, Close Solution

Данная команда меню закрывает текущее решение. Если текущий проект содержал несохраненную информацию, на дисплее появится окно сообщения, аналогичное изображенному на рис. П.15.

Рис. П.15. Окно сообщения о несохраненных файлах

Команда File, Save (<Ctrl>+<S>)

Выбор команды меню **File, Save** позволяет сохранить в файл изменения, внесенные в окно, которое имеет на настоящий момент фокус ввода. Если с окном, имеющим в настоящий момент фокус ввода, не связан никакой файл, то вместо данной команды автоматически вызывается команда **File, Save As**. Если в данный момент ни одно из окон редактирования файлов не имеет фокус ввода, то данная команда меню и соответствующая ей кнопка панели инструментов становятся недоступными.

Команда File, Save As

Выбор этой команды меню позволяет сохранить изменения, внесенные в окно, которое имеет на настоящий момент фокус ввода, в новом файле, сохранив при этом содержимое файла, с которым было связано данное окно, без изменений. При выборе этой команды появляется диалоговое окно **Save File As**, изображенное на рис. П.16.

Рис. П.16. Диалоговое окно **Save File As**

Если в этот момент ни одно из окон редактирования файлов не имеет фокуса ввода, то данная команда меню и соответствующая ей кнопка панели инструментов становятся недоступными.

Команда File, Advanced Save Options

При выборе данной команды меню появляется диалоговое окно **Advanced Save Options**, изображенное на рис. П.17.

Рис. П.17. Диалоговое окно **Advanced Save Options**

В раскрывающемся списке **Encoding:** этого диалогового окна можно выбрать кодировку, используемую при сохранении файла, а в раскрывающемся списке **Line endings:** — набор символов, помещаемый в конец каждой строки текста (зависит от используемой операционной системы).

Команда File, Save All(<Ctrl>+<Shift>+<S>)

Выбор данной команды меню позволяет сохранить изменения, внесенные во все открытые файлы проекта. Рекомендуется выбирать эту команду каждые 10-15 минут, если содержимое этих файлов не сохранялось другим способом. По умолчанию все файлы проекта сохраняются при его компиляции.

Команда File, Source Control

Данная команда меню используется для контроля исходного кода приложения. Контроль исходного кода приложения используется при совместной разработке приложения несколькими разработчиками. Поскольку в книге этот вопрос не рассматривался, описание команды опускается.

Команда File, Page Setup

Данная команда меню выводит на экран диалоговое окно **Page Setup**, изображенное на рис. П.18.

Рис. П.18. Диалоговое окно **Page Setup**

Это диалоговое окно позволяет задавать размер листа бумаги, на котором будет производиться распечатка, источник бумаги, размещение текста на нем и отступы текста от краев. Результат установок можно оценить в специальном окне предварительного просмотра текста.

Нажатие на кнопку **Printer...** позволит выбрать принтер, на котором будет произведена распечатка. В данном случае распечатка будет послана по факсу.

Команда File, Print (<Ctrl>+<P>)

Выбор данной команды меню приводит к инициализации процесса печати содержимого окна, которое на данный момент имеет фокус ввода. Если в текущий момент ни одно из окон редактирования файлов не имеет фокус ввода, то данная команда становится недоступной.

Непосредственно после выбора команды появляется диалоговое окно **Print**, изображенное на рис. П.19.

Рис. П.19. Диалоговое окно **Print**

В раскрывающемся списке **Name:** может быть выбран принтер, на котором будет произведена распечатка (если в системе установлен более чем один принтер).

Группа переключателей **Print range** (диапазон печати) позволяет определить диапазон печати в том случае, если в активном окне выделен блок. В этом случае при установке данного переключателя в положение **All** производится печать всего файла. При установке переключателя в положение **Selection** печатается только выделенный текст.

Нажатие кнопки **Properties** вызывает стандартное диалоговое окно **Properties**, позволяющее настроить используемый для печати принтер. Для печати документа нажмите кнопку **OK**.

Команда File, Recent Files

При выборе данной команды меню появляется контекстное меню, содержащее список последних закрытых файлов. Выбор имени файла в этом списке позволяет открыть его окно редактирования.

Команда File, Recent Projects

При выборе данной команды меню появляется контекстное меню, содержащее список последних открытых в данной системе программирования проектов (хотя, судя по расширениям, здесь речь идет не о проектах, а о решениях). Под номером **1** всегда стоит активный проект. Выбор имени проекта в этом списке позволяет открыть его для работы вместо активного в данный момент проекта.

Команда File, Exit

Выбор команды меню **File, Exit** завершает работу в среде программирования Visual Studio.NET.

Меню Edit

Меню **Edit**, приведенное на рис. П.20, содержит команды, осуществляющие операции редактирования, включая копирование, удаление, восстановление измененной информации, поиск и перемещение по файлу.

Рис. П.20. Меню **Edit**

Команда Edit, Undo (<Ctrl>+<Z>)

Выбор данной команды меню отменяет операции редактирования, выполненные ранее. Большая часть операций, таких как редактирование и удаление текста, могут отменяться. Если данная команда меню недоступна, это означает, что или пользователь не произвел ни одной операции редактирования с момента последнего сохранения файла, или все операции, которые

могут быть отменены, уже отменены, или последняя операция редактирования не может быть отменена.

В панели инструментов **Standard** около кнопки **Undo** находится треугольная кнопка, раскрывающая стек операций (список в обратном порядке от самых свежих до самых поздних операций редактирования, которые могут быть отменены). Раскрытый стек операций показан на рис. П.21. При отмене операций с использованием стека необходимо выделить последнюю отменяемую операцию в списке (расположенные выше операции отмечаются автоматически). Все операции, расположенные в стеке выше выделенной операции, и сама выделенная операция будут отменены.

Рис. П.21. Раскрытый стек операций

Команда Edit, Redo (<Ctrl>+<Y>)

Как только пользователь отменил некоторую операцию с использованием команды **Undo**, становится доступной команда **Redo**. Эта команда служит для отмены действия команды **Undo**. Около кнопки **Redo** в панели инструментов **Standard** также расположена треугольная кнопка, раскрывающая стек операций, в который записаны все операции, отмененные командой **Undo**.

Команда Edit, Cut (<Ctrl>+<X>)

Выбор данной команды меню приводит к копированию содержимого выделенного фрагмента в буфер обмена (Clipboard) и удалению этого фрагмента из исходного текста. Если в этот момент в активном окне отсутствует выделенный фрагмент, то команда меню **Cut** и соответствующая ей кнопка панели инструментов становятся недоступными.

Команда Edit, Copy (<Ctrl>+<C>)

Выбор данной команды меню приводит к копированию содержимого выделенного фрагмента в буфер обмена (Clipboard). В исходном файле не производится никаких изменений. Если в данный момент в активном окне отсутствует выделенный фрагмент, то эта команда меню и соответствующая ей кнопка панели инструментов становятся недоступными.

Команда Edit, Paste (<Ctrl>+<V>)

Выбор данной команды меню приводит к копированию содержимого буфера обмена (Clipboard) в окно, которому принадлежит в настоящее время фокус ввода в позицию, определяемую текстовым курсором. Если в окне имеется выделенный фрагмент, то его содержимое заменяется на содержимое буфера обмена и выделение снимается. Если в данный момент буфер обмена не содержит никакой информации, то эта команда меню и соответствующая ей кнопка панели инструментов становятся недоступными.

Команда Edit, Delete (<Del>)

Выбор данной команды меню удаляет выделенный текст или элемент. Если операция удаления восстановима, то эта операция добавляется в стек операций кнопки **Undo** в панели инструментов **Standard**.

Команда Edit, Select All (<Ctrl>+<A>)

Выбор этой команды меню выделяет весь текст файла, связанного с окном, имеющим в настоящее время фокус ввода. В окне редактирования текста программы выделяется весь текст содержащегося в нем файла. В окне редактирования ресурса диалогового окна выделяются все элементы управления данного диалогового окна.

Для выделения элементов управления диалогового окна можно, удерживая нажатой клавишу **<Ctrl>**, щелкнуть левой кнопкой мыши по всем элементам управления, которые нужно выделить, или выбрать команду меню **Edit**, **Select All** и, удерживая нажатой клавишу **<Ctrl>,** щелкнуть левой кнопкой мыши по всем элементам управления, которые нужно исключить из выделения.

Команда Edit, Find and Replace

Данная команда меню не является терминальной, а при ее выборе появляется контекстное меню, содержащее команды поиска и замены. Это контекстное меню показано на рис. П.22.

Рис. П.22. Контекстное меню **Find and Replace**

При выборе команды **Edit**, **Find and Replace**, **Find** или нажатии комбинации клавиш **<Ctrl>+<F>** появляется диалоговое окно **Find**, изображенное на рис. П.23.

Рис. П.23. Диалоговое окно **Find**

Это диалоговое окно используется для поиска текста в проекте. Поиск может быть произведен:

- только в текущем документе (положение переключателя **Current document**);
- во всех открытых документах (положение переключателя **All open documents**);
- только в выделенном фрагменте (положение переключателя **Selection only**);
- в некоторой области, определяемой контекстом (классе, функции и т.д.) (положение переключателя **Only:**).

Искомый текст помещается в текстовое поле раскрывающегося списка **Find what:**. В этом раскрывающемся списке хранятся фрагменты текста, по которым производился поиск в последнее время. Если искомый фрагмент текста содержится в раскрывающемся списке, его можно выбрать, а не вводить заново.

В диалоговом окне **Find** могут быть установлены следующие флажки:

- **Match case** — при сравнении учитывается регистр символов, используемый при написании текста в текстовом поле раскрывающегося списка **Find what:**. Если этот флажок сброшен, то при сравнении регистр символов не учитывается, что удобно при поиске идентификаторов, содержащих заглавные буквы в середине слова.
- **Match whole word** — устанавливает режим при котором текст в текстовом поле раскрывающегося списка **Find what:** рассматривается при сравнении как целое слово, и при поиске будут пропущены слова, в которых данный текст является составной частью другого слова.
- **Search hidden text** — производится поиск скрытого текста.
- **Search up** — поиск производится по направлению к началу текста.
- **Use:** — делает доступным связанный с ним раскрывающийся список. При выделении в этом списке строки **Regular expressions** в процессе поиска содержимое текстового поля раскрывающегося списка **Find what:** рассматривается как регулярное выражение. Под регулярным выражением понимается некоторый текст, включающий в себя специальные символы, такие как конец строки, любое число или другие символы. Для ввода специального символа достаточно нажать на кнопку со стрелкой, расположенную справа от текстового поля раскрывающегося списка **Find what:**, и выбрать соответствующую команду в появившемся контекстном меню, изображенном на рис. П.24.

  При выделении в этом списке строки **Wildcards** некоторые символы и последовательности символов будут рассматриваться как указатели на класс символов или последовательность символов. Так, например, вопросительный знак (?) будет обозначать любой символ, а звездочка (*) будет обозначать один или несколько символов.

Рис. П.24. Контекстное меню для ввода специальных символов

Обычно при работе с диалоговым окном **Find** в текстовое поле раскрывающегося списка **Find what:** вводится некоторый текст и нажимается кнопка **Find Next** (Найти следующий). Диалоговое окно **Find** закрывается, и в окне редактирования файла выделяется следующее вхождение данного текста. Для поиска нужной строки в тексте нужно последовательно открывать диалоговое окно **Find** и нажимать в нем кнопку **Find Next**. Эту процедуру нельзя назвать очень удобной.

Другой возможностью осуществить поиск текста в файле является использование кнопки **Mark All**. В этом случае у каждой строки, содержащей искомый текст, будет поставлена закладка. После этого можно просмотреть все строки, нажимая клавишу **<F2>**.

После первого же поиска, осуществленного с использованием диалогового окна **Find**, в текстовом поле раскрывающегося списка **Find**, расположенного в панели инструментов **Standard**, появится текст, введенный в текстовое поле раскрывающегося списка **Find what:**. При выделении данного текста и нажатии клавиши **<Enter>** будет производиться поиск в направлении конца файла.

При выборе команды **Edit**, **Find and Replace**, **Replace** или нажатии комбинации клавиш **<Ctrl>+<H>** появляется диалоговое окно **Replace**, изображенное на рис. П.25.

Рис. П.25. Диалоговое окно **Replace**

Это диалоговое окно, во многом аналогичное диалоговому окну **Find**, используется для замены найденного текста новым. Использование текстового поля раскрывающегося списка **Find what:** и флажков **Match case**, **Match whole word**, **Search hidden text**, **Search up, Use:** и связанного с ним раскрывающегося списка, а также переключателей группы **Search** в диалоговом окне **Replace** ничем не отличается от использования этих элементов управления в диалоговом окне **Find**. Основным преимуществом диалогового окна **Replace** перед диалоговым окном **Find** является то, что при нажатии кнопки **Find Next** данное окно не закрывается, и это позволяет существенно упростить процедуру поиска информации в файле.

В текстовом поле раскрывающегося списка **Replace with:** помещается текст, на который следует заменить искомый текст. Так же как и в случае с раскрывающимся списком **Find what:**, в этом раскрывающемся списке хранятся фрагменты текста, использовавшиеся в предыдущих заменах. Если требуемый текст содержится в раскрывающемся списке, то его можно выбрать из него, а не вводить вручную. Текст в текстовом поле раскрывающегося списка **Replace with:** может представлять собой регулярное выражение.

После того как в тексте файла с использованием кнопки **Find Next** будет найден требуемый фрагмент текста, пользователь может нажать кнопку **Replace**. В результате чего найденный фрагмент текста будет заменен и будет произведен поиск следующего фрагмента текста. Если пользователь нажал кнопку **Replace All**, то операция поиска и замены текста по всему файлу будет произведена автоматически без участия пользователя. Кнопку **Replace All** следует использовать с особой осторожностью, так как при неправильном выборе установок в окне результаты быстрой замены придется исправлять в течение нескольких часов, поскольку стек операций команды **Undo** имеет ограниченные размеры. Если вы все-таки решились использовать эту кнопку, то настоятельно рекомендуется по крайней мере для первых замен использовать кнопку **Replace** и убедиться, что при замене не возникает никаких проблем.

При выборе команды **Edit**, **Find and Replace**, **Find in Files** или нажатии комбинации клавиш **<Ctrl>+<Shift>+<F>** появляется диалоговое окно **Find in Files**, изображенное на рис. П.26.

Рис. П.26. Диалоговое окно **Find in Files**

Это диалоговое окно используется для поиска текста во всех файлах проекта, расширения которых указаны в текстовом поле раскрывающегося списка **File types:**. Так же как и в диалоговом окне **Find** искомый текст помещается в текстовое поле раскрывающегося списка **Find what:**. Работа с ним аналогична работе с таким же раскрывающимся списком в диалоговом окне **Find**.

В текстовом поле раскрывающегося списка **Look in:** указывается область поиска. Это может быть текущий документ, выделенный фрагмент, текущий проект и другие области. Область поиска может быть настроена при нажатии на кнопку с тремя точками, расположенную справа от данного раскрывающегося списка. При нажатии на эту кнопку появляется диалоговое окно **Look In**, изображенное на рис. П.27.

Рис. П.27. Диалоговое окно **Look In**

В диалоговом окне **Find In Files** могут быть установлены флажки **Match case**, **Match whole word** и **Use:**, назначение которых совпадает с назначением одноименных флажков в диалоговых окнах **Find** и **Replace**. Кроме того, в данном диалоговом окне могут быть установлены следующие флажки:

- **Display in Find 2** — собранная информация отображается в окне **Find Results 2**.
- **Display file names only** — в окне **Find Results** будут выводиться только имена файлов, в которых найден указанный текст (то есть не будет выводиться номер и содержимое строки, в которой найден этот текст).

После ввода всей необходимой информации и нажатия кнопки **Find** среда программирования Visual Studio.NET производит поиск в файлах данного проекта и выводит его результаты в окно **Find Results**, как это показано на рис. П.28. Если окно **Find Results** было до этого скрыто, то оно будет выведено на экран. Номер окна, в которое будут выведены результаты поиска, зависит от состояния флажка **Display in Find 2**.

Рис. П.28. Результат поиска в файлах проекта

Чтобы перейти к найденной строке в файле достаточно дважды щелкнуть по соответствующей ей строке в окне **Find Results**. При этом окно, содержащее

данную строку, получит фокус ввода, и текстовый курсор переместится в начало этой строки в окне редактирования. Если соответствующее окно редактирования не было открыто, оно автоматически открывается в текстовом режиме. Поэтому поиск в файлах может использоваться для открытия файла ресурсов в текстовом режиме. Для этого нужно произвести поиск по одному из идентификаторов ресурсов, а затем дважды щелкнуть левой кнопкой мыши по строке, расположенной в файле ресурсов.

При нажатии кнопки **Replace** в диалоговом окне **Find in Files** оно превращается в диалоговое окно **Replace in Files**, изображенное на рис. П.29. Это диалоговое окно можно также вызвать командой меню **Edit**, **Find and Replace**, **Replace in Files** или нажатием комбинации клавиш **<Ctrl>+<Shift>+<H>**.

Рис. П.29. Диалоговое окно **Replace in Files**

Это диалоговое окно работает аналогично диалоговому окну **Replace**, но позволяет произвести замену текста не только в окне, имеющем в настоящее время фокус ввода, но и во всех файлах, определяемых содержимым текстовых полей раскрывающихся списков **Look in:** и **File types:**.

При выборе команды **Edit**, **Find and Replace**, **Find Symbol** или нажатии комбинации клавиш **<Ctrl>+<Shift>+<Y>** появляется диалоговое окно **Find Symbol**, изображенное на рис. П.30.

Рис. П.30. Диалоговое окно **Find Symbol**

Это окно позволяет производить поиск символьных объектов, под которыми понимаются именованные области, классы, структуры, интерфейсы, типы и их члены, такие как свойства, методы, события, переменные и константы. Результаты поиска выводятся в окно **Find Symbol Results**, изображенное на рис. П.31.

Рис. П31. Окно **Find Symbol Results**

Диалоговое окно **Find Symbol** имеет следующие элементы управления:

- **Find what:** — текстовое поле раскрывающегося списка, в котором указывается полное или частичное имя искомого символа. В раскрывающемся списке содержатся символы, поиск по которым уже производился.
- **Look in:** — текстовое поле раскрывающегося списка, в котором указывается область, в которой следует производить поиск. В этом раскрывающемся списке может быть выбрана одна из следующих областей поиска:
  - **Active Project** — ограничивает область поиска активным проектом и его компонентами, на которые имеются ссылки.
  - **Selected Components** — позволяет пользователю самому определить область поиска, которая может включать в себя проекты, их компоненты, на которые имеются ссылки, и другие внешние компоненты.
- Для выбора области поиска пользователь должен нажать кнопку **Browse** (кнопку **...**). При этом появится диалоговое окно **Selected Components**, изображенное на рис. П.32. Для включения компонента в область поиска необходимо раскрыть папку, в которой он содержится, и установить флажок у его имени.
- **Match whole word** — при установке переключателя в это положение будет произведен поиск символов, полностью совпадающих с указанной строкой. Например, при поиске по строке "Demo" будет найден символ "Demo", но не будут найдены символы "DemoObject" и "CDemo".

Рис. П.32. Диалоговое окно **Selected Components**

- **Match prefix** — при установке переключателя в это положение будет произведен поиск символов, начинающихся с указанной строки. То есть не будут найдены символы, в которых перед указанной строкой стоит некоторый префикс. Например, при поиске по строке "Demo" будут найдены символы "Demo" и "DemoObject", но не будет найден символ "CDemo".
- **Match substring** — при установке переключателя в это положение будет произведен поиск символов, включающих указанную строку. Например, при поиске по строке "Demo" будут найдены символы "Demo", "DemoObject" и "CDemo".
- **Match case** — полностью аналогичен одноименному флажку в диалоговых окнах **Find** и **Replace**. При поиске учитывается регистр символов.

Команда Edit, Go To Line (<Ctrl+G>)

Эта команда меню выводит на экран диалоговое окно **Go To Line**, изображенное на рис. П.33.

Рис. П.33. Диалоговое окно **Go To Line**

Диалоговое окно **Go To Line** позволяет перейти к строке с конкретным номером, который указывается в текстовом поле данного диалогового окна.

Для того чтобы пользователь не ввел в него недопустимый номер строки, в заголовке текстового поля указан диапазон допустимых значений.

Команда Edit, Insert File As Text

Данная команда меню используется для вставки содержимого одного файла в другой в текстовом виде. Вставка производится в текущую позицию текстового курсора, а для выбора вставляемого файла на экран выводится диалоговое окно **Insert File**, изображенное на рис. П.34.

Рис. П.34. Диалоговое окно **Insert File**

Команда Edit, Advanced

Данная команда меню не является терминальной. При ее вызове на экране появляется контекстное меню, содержащее набор команд форматирования текста, как это показано на рис. П.35.

Контекстное меню **Advanced** содержит следующие команды:

- **Format Selection** (Выбор формата) — выбор данной команды производит форматирование выделенного текста в соответствии с требованиями, предъявляемыми при написании программ.
- **Tabify Selection** (Табулирование) — преобразует последовательность пробелов в выделенном фрагменте текста в символы табуляции.
- **Untabify Selection** (Растабулирование) — преобразует в выделенном фрагменте текста символы табуляции в последовательность пробелов.

Рис. П.35. Контекстное меню **Advanced**

- **Make Selection Uppercase** (Перевод в верхний регистр) — преобразует символы выделенного текста в заглавные.
- **Make Selection Lowercase** (Перевод в нижний регистр) — преобразует символы выделенного текста в строчные.
- **Delete Horizontal White Space** (Удалить горизонтальные пробелы) — уничтожает множественные пробелы и символы табуляции в позиции текстового курсора. В результате выполнения этой операции на месте нескольких пробелов остается один пробел. На месте символов табуляции или комбинации этих символов с пробелами не остается ничего.
- **View White Space** (Визуализация пробелов) — вставляет символы-заполнители (".” для пробела и ">>" для символа табуляции).
- **Word Wrap** (Автоматический переход на новую строку) — при установке этого флажка в окне редактирования текста отменяется режим горизонтальной прокрутки. При достижении текстом правой границы окна он автоматически переносится на новую строку.
- **Incremental Search** (Поиск по возрастанию) — эта команда меню позволяет осуществить более быстрый поиск, чем диалоговое окно **Find**. При

выборе этой команды внешне ничего не происходит, однако, при вводе с клавиатуры символов в тексте начинает выделяться первое включение данной комбинации символов, начиная с текущей позиции курсора. По мере ввода новых символов выделенный фрагмент изменяется и, возможно, перемещается по тексту. Эта команда удобна для поиска первого вхождения идентификатора.

- **Increase Line Indent** (Увеличить отступ) — эта команда перемещает текущую строку на одну позицию табуляции вправо.
- **Decrease Line Indent** (Уменьшить отступ) — эта команда перемещает текущую строку на одну позицию табуляции влево.

Как следует из рис. П.35, многим командам этого меню соответствуют двойные нажатия комбинаций клавиш. То есть для вызова соответствующей команды необходимо последовательно нажать две различные (или одинаковые) комбинации клавиш. Надо полагать, что разработчиков Visual Studio.NET замучила ностальгия по Turbo Pascal и другим продуктам десятилетней давности.

Команда Edit, Bookmarks

Эта команда меню не является терминальной. При ее вызове на экране появляется контекстное меню, содержащее набор команд для работы с закладками, как это показано на рис. П.36.

Контекстное меню **Bookmarks** содержит следующие команды:

- **Toggle Bookmark** (Переключить закладку) — выбор данной команды позволяет установить закладку в строке, если она в ней отсутствует, и удалить эту закладку, если она присутствует.
- **Next Bookmark** (Следующая закладка) — выбор данной команды позволяет перейти к следующей закладке в данном окне.
- **Previous Bookmark** (Предыдущая закладка) — выбор данной команды позволяет перейти к предыдущей закладке в данном окне.
- **Clear Bookmarks** (Удалить закладки) — выбор данной команды позволяет удалить все закладки в данном окне.
- **Add Task List Shortcut** (Добавить метку списка задач) — строка, помеченная такой меткой, выводится в окно **Task List** (если в контекстном меню данного окна была выбрана команда **Show Tasks**, **Shortcut** или **Show Tasks**, **All**). Двойной щелчок левой кнопкой мыши на этой строке в окне **Task List** позволяет перейти к ней в окне редактирования. Если в текущей строке уже установлена эта метка, эта команда в меню заменяется командой **Remove Task List Shortcut** (Удалить метку списка задач).

Рис. П.36. Контекстное меню **Bookmarks**

Команда Edit, Outlining

Данная команда меню не является терминальной. При ее вызове на экране появляется контекстное меню, содержащее набор команд для работы со структурой проекта, как это показано на рис. П.37.

Контекстное меню **Outlining** содержит следующие команды:

- **Hide Selection** (Скрыть выделение) — выбор данной команды позволяет скрыть выделенный фрагмент программы. При этом весь выделенный фрагмент заменяется рамкой с тремя точками, как показано на рис.П.38. Обратите внимание на то, что выделенный фрагмент может не являться законченной конструкцией.
- **Toggle Outlining Expansion** (Переключить режим отображения) — если в текущей строке расположен текст программы, эта команда скрывает элемент программы, которому принадлежит данный программный текст(метод, класс, именованная область и т. д.), как это показано на рис. П.39. Если текущая строка уже была скрыта, команда отменит скрытие и выведет программный текст.

Рис. П.37. Контекстное меню **Outlining**

Рис. П.38. Скрытие выделенного фрагмента

Рис. П.39. Скрытие элемента программы

Рис. П.40. Полное скрытие элементов программы

- **Toddle All Outlining** (Переключить режим структуры) — если текущая строка содержит текст программы, команда скрывает весь программный текст окна в структурах высшего уровня, как это показано на рис. П.40. В противном случае эта команда отменяет скрытие всех элементов в текущем окне.
- **Stop Outlining** (Отмена режима структуры) — отменяет скрытие всех элементов в окне и отменяет действие всех перечисленных выше команд данного контекстного меню (команды остаются доступными, но их вызов не приводит ни к каким последствиям).
- **Stop Hiding Current** (Отменить текущее скрытие) — отменяет действие команды **Hide Selection**.
- **Collapse to Definitions** (Сжать до объявлений) — выводит структуру программы в текущем окне, как это показано на рис. П.41.

Рис. П.41. Структура программы

Команда Edit, IntelliSense

Данная команда меню не является терминальной. При ее вызове на экране появляется контекстное меню, содержащее набор команд, позволяющих пользователю автоматизировать процесс написания программы, как это по-

казано на рис. П.42. Большинство возможностей, реализованных в этом меню, выполняются автоматически, поэтому его следует вызывать только в крайних случаях.

Рис. П.42. Контекстное меню **IntelliSense**

Контекстное меню **IntelliSense** содержит следующие команды:

- **List Members** (Список членов) — выводит список членов типа для ближайшего к текстовому курсору объекта, как это показано на рис. П.43.
- **Parameter Info** (Сведения о формальных параметрах) — при нахождении текстового курсора в списке аргументов метода выводит информацию о данном формальном параметре и о самом методе, как это показано на рис. П.44.
- **Quick Info** (Быстрая справка) — позволяет получить краткую справку по элементу программы, как это показано на рис. П.45.
- **Complete Word** (Завершить слово) — позволяет автоматически завершить ввод идентификатора, если введенной информации достаточно для его однозначного определения.

Рис. П.43. Список членов объекта

Рис. П.44. Информация о формальных параметрах метода

Рис. П.45. Краткая справка по переменной

Меню View

Меню **View**, приведенное на рис. П.46, содержит команды, осуществляющие управление способом отображения информации на экране, включая управление отображением панелей инструментов и окон среды разработки.

Команда View, Code (<F7>)

Команда меню **Code** позволяет открыть окно редактирования программного кода, выделенного в окне **Solution Explorer** или **Class View**.

Команда View, Open

Команда **Open** открывает указанный файл. При работе с файлами программ полностью аналогична команде **View**, **Code**.

Команда View, Open With

Данная команда позволяет выбрать редактор, в котором будет открыт указанный файл. Эта же команда позволяет изменить редактор, используемый в текущем окне. В последнем случае появится окно сообщения, предлагающее закрыть текущее окно. При выборе этой команды на экран выводится диалоговое окно **Open With**, изображенное на рис. П.13.

Рис. П.46. Меню **View**

Команда View, Solution Explorer (<Ctrl>+<Alt>+<L>)

Данная команда открывает окно **Solution Explorer**, содержащее список файлов проекта. Это окно, как правило, выводится в виде вкладки, но на рис. П.47 оно представлено как самостоятельное окно.

Рис. П.47. Окно **Solution Explorer**

Команда View, Class View (<Ctrl>+<Shift>+<C>)

Данная команда открывает окно **Class View**, которое содержит иерархическую структуру типов и именованных областей проекта. Это окно, как правило, выводится в виде вкладки, но на рис. П.48 окно **Class View** представлено как самостоятельное окно.

Рис. П.48. Окно **Class View**

Команда View, Server Explorer (<Ctrl>+<Alt>+<S>)

Данная команда открывает вкладку **Server Explorer**, как это показано на рис. П.49, которая в свою очередь позволяет задать связь приложения с базами данных и серверами.

Команда View, Resource View (<Ctrl>+<Shift>+<E>)

Данная команда открывает окно **Resource View**, которое содержит иерархический список ресурсов приложения. Поскольку приложения C#, как правило, напрямую не работают с ресурсами, это окно в большинстве случаев пусто.

Команда View, Properties Window (<F4>)

Данная команда открывает окно **Properties**, содержащее набор свойств того или иного программного элемента. Это окно позволяет разработчику вносить изменения в эти свойства. Это окно, как правило, выводится в виде вкладки, но на рис. П.50 оно представлено как самостоятельное окно.

Рис. П.49. Вкладка **Server Explorer**

Рис. П.50. Окно **Properties**

Команда View, Toolbox (<Ctrl>+<Alt>+<X>)

Данная команда открывает вкладку **Toolbox**, как это показано на рис. П.51, которая позволяет автоматически создавать в приложении некоторые объекты.

Рис. П.51. Вкладка **Toolbox**

Команда View, Web Browser (<Ctrl>+<Alt>+<R>)

Данная команда открывает окно Web-браузера. В данном случае это рассмотренная выше первая страница Visual Studio.NET.

Команда View, Other Windows

Данная команда меню не является терминальной и при ее выборе открывается контекстное меню, изображенное на рис. П.52.

Открываемые в нем окна используются для вывода различной информации в процессе разработки и отладки приложения. Как правило, они открываются автоматически, если в них возникает необходимость.

Команда View, Show Tasks

Данная команда меню не является терминальной и при ее выборе открывается контекстное меню, изображенное на рис. П.53.

Рис. П.52. Контекстное меню **Other Windows**

Рис. П.53. Контекстное меню **Show Tasks**

Команды этого меню определяют информацию, выводимую в окно **Task List**, позволяют выполнять некоторые операции, необходимые при работе с несколькими заданиями.

Команда View, Toolbars

Данная команда меню не является терминальной и при ее выборе открывается контекстное меню, изображенное на рис. П.54.

Рис. П.54. Контекстное меню **Toolbars**

Команды данного контекстного меню являются, по сути дела, флажками, определяющими набор выводимых среде разработки панелей инструментов.

Команда View, Full Screen (<Shift>+<Alt>+<Enter>)

Выбор данной команды меню скрывает все панели инструментов, меню, окна вывода и панель задач **Windows**, предоставляя пользователю возможность работать в полноэкранном режиме, как это показано на рис. П.55.

Рис. П.55. Работа в полноэкранном режиме

Для восстановления нормального режима работы в окне остается панель инструментов **Full Screen**, содержащая единственную кнопку **Full Screen**, на которую и нужно нажать для выхода из полноэкранного режима отображения.

Команда View, Server Explorer View

Данная команда меню не является терминальной и при ее выборе открывается контекстное меню, изображенное на рис. П.56.

Это меню позволяет сохранить текущие настройки вкладки **Server Explorer**. При выборе команды **Save Current View** сохранение производится под тем же именем, под которым они были загружены, а при выборе команды **Save View As...** появляется диалоговое окно **Save View**, изображенное на рис. П.57. Это окно позволяет задать новое имя для данной конфигурации настроек.

Команда View, Navigate Backward

Как уже говорилось, при рассмотрении кнопок панели инструментов **Standard**, в которой дублирована данная команда меню, ее выбор позволяет отменять команды перемещения курсора между реализациями различных типов.

Рис. П.56. Контекстное меню **Server Explorer View**

Рис. П.57. Диалоговое окно **Save View**

Команда View, Navigate Forward

Данная команда меню отменяет однократное действие команды **Navigate Backward**.

Команда View, Property Pages

Данная команда меню позволяет открыть диалоговое окно **Property Pages**, изображенное на рис. П.58. В этом окне могут быть изменены установки текущего проекта. Чтобы эта команда меню стала доступной, следует выделить имя проекта в окне **Solution Explorer** или **Class View**.

Рис. П.58. Диалоговое окно **Property Pages**

Меню Project

Меню **Project**, приведенное на рис. П.59, содержит команды, осуществляющие управление разрабатываемым проектом в целом.

Рис. П.59. Меню **Project**

Команда Project, Add Windows Form

При выборе данной команды меню появляется диалоговое окно **Add New Item**, изображенное на рис. П.60. В данном случае это диалоговое окно используется для создания нового формализованного приемника Windows и добавления его в проект.

Рис. П.60. Диалоговое окно **Add New Item**

Команда Project, Add Inherited Form

При выборе данной команды меню появляется диалоговое окно **Add New Item**, изображенное на рис. П.60. В данном случае это диалоговое окно используется для создания нового унаследованного формализованного приемника Windows и добавления его в проект.

Команда Project, Add User Control

При выборе данной команды меню появляется диалоговое окно **Add New Item**, изображенное на рис. П.60. В данном случае это диалоговое окно используется для создания нового пользовательского элемента управления и добавления его в проект.

Команда Project, Add Inherited Control

При выборе данной команды меню появляется диалоговое окно **Add New Item**, изображенное на рис. П.60. В данном случае это диалоговое окно используется для создания нового унаследованного пользовательского элемента управления и добавления его в проект.

Команда Project, Add Component

При выборе данной команды меню появляется диалоговое окно **Add New Item**, изображенное на рис. П.60. В данном случае это диалоговое окно используется для создания нового компонента Windows и добавления его в проект.

Команда Project, Add Class

При выборе данной команды меню появляется диалоговое окно **C# Class Wizard**, изображенное на рис. П.61, позволяющее создать новый класс, поместить его в соответствующую именованную область и выбрать для него модификаторы. Однако оно почему-то не позволяет поместить этот класс в один из уже существующих файлов и выводит при этом целую серию окон сообщения с предупреждением о том, что созданное Visual Studio.NET имя файла, от которого отказался пользователь, недопустимо.

Рис. П.61. Диалоговое окно **C# Class Wizard**

Команда Project, Add Method

Данная команда появляется в меню **Project** в том случае, если в окне **Class View** выделен идентификатор класса. При выборе данной команды появляется диалоговое окно **C# Method Wizard**, изображенное на рис. П.62, которое позволяет создать новый метод, задать его параметры и поместить его в класс, чей идентификатор выделен в окне **Class View**.

Рис. П.62. Диалоговое окно **C# Method Wizard**

Команда Project, Add Property

Данная команда появляется в меню **Project** в том случае, если в окне **Class View** выделен идентификатор класса. При выборе данной команды появляется диалоговое окно **C# Property Wizard**, изображенное на рис. П.63, которое позволяет создать новое свойство, определить процедуры доступа к нему, задать его параметры и поместить в класс, чей идентификатор выделен в окне **Class View**.

Рис. П.63. Диалоговое окно **C# Property Wizard**

Команда Project, Add Field

Данная команда появляется в меню **Project** в том случае, если в окне **Class View** выделен идентификатор класса. При выборе данной команды появляется диалоговое окно **C# Field Wizard**, изображенное на рис. П.64, которое позволяет создать новое поле или константу, задать его параметры и поместить в класс, чей идентификатор выделен в окне **Class View**.

Рис. П.64. Диалоговое окно **C# Field Wizard**

Команда Project, Add Indexer

Данная команда появляется в меню **Project** в том случае, если в окне **Class View** выделен идентификатор класса. При выборе данной команды появляется диалоговое окно **C# Indexer Wizard**, изображенное на рис. П.65, которое позволяет создать новый индексатор, задать его параметры и поместить в класс, чей идентификатор выделен в окне **Class View**.

Команда Project, Add New Item (<Ctrl>+<Shift>+<A>)

При выборе данной команды меню появляется диалоговое окно **Add New Item**, изображенное на рис. П.60. Поскольку для основных объектов, которые могут быть созданы с использованием этого диалогового окна, существуют рассмотренные выше специальные команды меню, при вызове это окно устанавливается на создание класса C# и добавление его в проект. В данном случае отсутствует возможность выбора параметров создаваемого класса.

Рис. П.65. Диалоговое окно **C# Indexer Wizard**

Команда Project, Add Existing Item (<Ctrl>+<Alt>+<A>)

При выборе данной команды меню появляется диалоговое окно **Add Existing Item**, изображенное на рис. П.66.

Рис. П.66. Диалоговое окно **Add Existing Item**

Данное диалоговое окно позволяет добавить в проект все типы и именованные области, содержащиеся в выделенном файле. Подобный подход заставляет разработчика семь раз отмерить, прежде, чем поместить описание двух типов в один файл. Кроме того, это затрудняет использование в проектах имен файлов, создаваемых Visual Studio.NET. Поскольку эти имена не содержат имен приложения и формируются единообразно, появляется большая вероятность того, что в данном приложении уже существует файл с тем же именем, что и добавляемый. В этом случае Visual Studio.NET не находит лучшего выхода, чем заменить существующий файл новым.

Команда Project, Exclude From Project

Данная команда меню позволяет исключить компонент (файл, выведенный в активном окне) из проекта. При этом активное окно закрывается, а соответствующий файл исключается из окна **Solution Explorer**.

Команда Project, Show All Files

При выборе данной команды меню в окне **Solution Explorer** выводятся не только редактируемые файлы проекта (в данном случае, имеющие расширение .cs), но и все остальные его файлы, включая промежуточные и исполняемые файлы. При повторном выборе этой команды меню традиционный набор выводимых файлов восстанавливается.

Команда Project, Add Reference

При выборе данной команды меню появляется диалоговое окно **Add Reference**, изображенное на рис. П.67.

Рис. П.67. Диалоговое окно **Add Reference**

Данное диалоговое окно позволяет добавить в проект ссылки на компоненты, перечисленные в его вкладках.

Команда Project, Add Web Reference

При выборе данной команды меню появляется диалоговое окно **Add Web Reference**, изображенное на рис. П.68.

Рис. П.68. Диалоговое окно **Add Web Reference**

Данное диалоговое окно позволяет добавить в проект ссылки на компоненты, доступные в Сети. При нахождении браузера на Web-странице, содержащей сервисы и наборы данных, ссылки на них отображаются в окне списка **Available references:**.

Команда Project, Set as StartUp Project

Выбор данной команды меню приводит к тому, что текущий проект будет автоматически загружаться при старте Visual Studio.NET.

Меню Build

Меню **Build**, приведенное на рис. П.69, содержит команды компиляции, компоновки и отладки.

Рис. П.69. Меню **Build**

Команда Build, Build (<Ctrl>+<Shift>+<B>)

Данная команда позволяет перекомпилировать все файлы, в которые были внесены изменения. На панели инструментов **Build** имеется кнопка, соответствующая данной команде.

Команда Build, Rebuild

Данная команда позволяет перекомпилировать все файлы проекта независимо от того, были в них внесены изменения после последней компиляции или нет. Иногда встречаются случаи, когда в процессе компиляции пропускается файл, который должен быть перекомпилирован. Обычно это имеет место в том случае, если в данный файл были внесены изменения в обход среды программирования Visual Studio.NET (например, вместо одного из файлов была записана его старая версия, взятая из архива).

Команда Build, Deploy

Данная команда меню используется для установки создаваемого компонента на удаленном компьютере. Для установки компонента необходимо создать специальный проект установки, в котором определяется состав комплекта поставки данного компонента и другие параметры его распространения.

Команда Build, Batch Build

Выбор данной команды меню выводит на экран диалоговое окно **Batch Build**, изображенное на рис. П.70.

Рис. П.70. Диалоговое окно **Batch Build**

Данное диалоговое окно позволяет произвести одновременную компиляцию нескольких конфигураций приложения (например, отладочной и распространяемой). Компилируемые конфигурации отмечаются в столбце **Build** окна списка **Check the project configurations to build:** (Отметьте компилируемые конфигурации).

Для компиляции измененных файлов нажимается кнопка **Build**, а для компиляции всех файлов проекта — кнопка **Rebuild**. После успешной компиляции производится компоновка файлов.

Команда Build, Configuration Manager

При выборе данной команды меню появляется диалоговое окно **Configuration Manager**, изображенное на рис. П.71.

Данное диалоговое окно позволяет выбрать компилируемые и распространяемые конфигурации создаваемого проекта, а также используемую в нем платформу.

Рис. П.71. Диалоговое окно **Configuration Manager**

Меню Debug

Меню **Debug**, приведенное на рис. П.72, содержит команды запуска приложения на исполнение и команды, используемые при его отладке.

Рис. П.72. Меню **Debug**

Команда Debug, Windows

Данная команда меню не является терминальной и при ее выборе появляется контекстное меню, изображенное на рис. П.73. При запуске приложения на исполнение в этом меню появляются дополнительные команды.

Рис. П.73. Контекстное меню **Windows**

Команды данного контекстного меню используются для вывода на экран окон, в которые будет выводиться различная отладочная информация.

Команда Debug, Start (<F5>)

Выбор данной команды меню запускает приложение на исполнение в режиме отладки. После запуска приложения на исполнение на месте этой команды меню появляется команда **Continue**, возобновляющая выполнение программы в режиме отладки до следующей точки останова или до конца программы, если при ее дальнейшем выполнении не встретятся точки останова.

Команда Debug, Break (<Ctrl>+<Alt>+<Break>)

Данная команда появляется в меню только в процессе отладки приложения. Ее выбор приостанавливает выполнение текущего приложения, оставляя возможность его старта с текущей точки.

Команда Debug, Stop Debugging (<Shift>+<F5>)

Данная команда появляется в меню только в процессе отладки приложения. Ее выбор завершает процесс выполнения отлаживаемого приложения.

Команда Debug, Detach All

Данная команда появляется в меню только в процессе отладки приложения. Ее выбор завершает процесс отладки приложения, не прекращая его работы. То есть от приложения отключаются все отладочные процессы.

Команда Debug, Restart (<Ctrl>+<Shift>+<F5>)

Данная команда появляется в меню только в процессе отладки приложения. Ее выбор перезапускает исполняемое приложение. Выполнение перезапущенного приложения останавливается на его первом исполняемом выражении.

Команда Debug, Apply Code Changes (<Alt>+<F10>)

Данная команда появляется в меню только в процессе отладки приложения и становится доступной, после того как в приложение будут внесены какие-либо изменения. Назначение этой команды остается загадкой, поскольку как до ее выбора, так и после него при попытке продолжить выполнение программы появляется окно сообщения, запрашивающее информацию о том, следует ли перекомпилировать проект, прежде чем продолжить его исполнение с текущей точки останова.

Команда Debug, Start Without Debugging (<Ctrl>+<F5>)

Выбор данной команды меню запускает приложение на исполнение в режиме распространяемой версии. В этом режиме приложение не реагирует на точки останова и не выдает отладочной информации.

Команда Debug, Processes

При выборе данной команды меню появляется диалоговое окно **Processes**, изображенное на рис. П.74.

Рис. П.74. Диалоговое окно **Processes**

Данное диалоговое окно позволяет получить информацию обо всех процессах, запущенных на всех компьютерах сети, и осуществить простейшее управление отлаживаемыми процессами (прервать или остановить их выполнение процесса, или же отключить его).

Команда Debug, Step Into (<F11>)

Выбор данной команды меню позволяет выполнить один оператор программы и снова остановиться. В том случае, если текущий оператор является функцией, доступной для отладки, следующим оператором программы является первый оператор данной функции (ее заголовок). Если текущий оператор не является функцией, данная команда работает аналогично команде меню **Debug**, **Step Over**. Если текст данной функции не содержится в директориях, указанных пользователем при настройке системы, система выводит диалоговое окно, предлагая указать путь к данному файлу. Если данная функция содержится в системной библиотеке и недоступна для отладки, система выводит окно сообщения с соответствующим текстом.

Команда Debug, Step Over (<F10>)

Выбор данной команды меню позволяет выполнить один оператор программы и снова остановиться независимо от того, является ли данный оператор функцией или нет.

Команда Debug, Step Out (<Shift>+<F11>)

Выбор данной команды меню позволяет выйти из текущей функции и остановиться на первом операторе программы, расположенном за вызовом данной функции.

Команда Debug, Step By

Данная команда меню не является терминальной и при ее выборе открывается контекстное меню, показанное на рис. П.75.

Рис. П.75. Контекстное меню **Step By**

Это меню позволяет выбрать шаг перемещения при выполнении команд **Debug**, **Step Into** и **Debug, Step Over**. В нем может быть выбрано одно из следующих значений шага:

- **Line** — будет осуществлен переход на следующую строку исходного кода программы.
- **Statement** — будет осуществлен переход к следующему выражению.
- **Instruction** — будет осуществлен переход на следующую инструкцию ассемблера.

Команда Debug, Set Next Statement

При выборе данной команды меню управление передается выражению, в котором расположен текстовый курсор. Это выражение должно располагаться в том же методе, что и текущая точка останова.

Команда Debug, Run to Cursor (<Ctrl>+<F10>)

Выбор данной команды меню позволяет продолжить выполнение программы и остановиться на операторе, на котором в настоящий момент расположен текстовый курсор.

Команда Debug, Go To Disassembly

При выборе данной команды меню открывается окно, содержащее ассемблерный код выполняемой программы, как это показано на рис. П.76.

Рис. П.76. Окно **Disassembly**

Команда Debug, QuickWatch (<Ctrl>+<Alt>+<Q>)

Выбор данной команды меню выводит на экран диалоговое окно **QuickWatch**, изображенное на рис. П.77.

Рис. П.77. Диалоговое окно **QuickWatch**

Данное диалоговое окно позволяет просмотреть значение выражения, которое было выделено в момент вызова данной команды меню или было введено пользователем в текстовое поле раскрывающегося списка **Expression:**. Нажатие кнопки **Add Watch** позволяет добавить это выражение в окно **Watch**.

Команда Debug, New Breakpoint (<Alt>+<F9>+<N>)

Выбор данной команды меню выводит на экран диалоговое окно **New Breakpoint**, изображенное на рис. П.78.

Данное диалоговое окно позволяет устанавливать точки останова, указывая при этом не только их местоположение, но и достаточно сложные условия остановки в них программы.

Команда Debug, Clear All Breakpoints (<Ctrl>+<Shift>+<F9>)

Вызов данной команды меню убирает из приложения все установленные в нем точки останова.

Команда Debug, Save As CrashDump

Выбор данной команды меню выводит на экран диалоговое окно **Save crashdump As...**, изображенное на рис. П.79.

Рис. П.78. Диалоговое окно **New Breakpoint**

Рис. П.79. Диалоговое окно **Save crashdump As...**

Данное диалоговое окно позволяет сохранить текущее состояние проекта в формате, используемом для определения причин его краха.

Команда Debug, Disable Breakpoint (<Ctrl>+<F9>)

Вызов данной команды меню отменяет действие точки останова, не убирая ее. При этом на месте этой команды появляется команда **Enable Breakpoint**, восстанавливающая действие точки останова. Если в текущей строке не была установлена точка останова, эта команда становится недоступной.

Меню Tools

Меню **Tools**, приведенное на рис. П.80, содержит команды настройки среды Visual Studio.NET и команды доступа к автономным утилитам.

Рис. П.80. Меню **Tools**

Команда Tools, Connect to Database

Данная команда меню позволяет подключить приложение к серверу SQL базы данных. При выборе данной команды раскрывается панель **Server Explorer** и появляется диалоговое окно **Data Link Properties**, как это показано на рис. П.81.

Рис. П.81. Диалоговое окно **Data Link Properties**

Команда Tools, Connect to Server

Данная команда меню позволяет подключить приложение к новому серверу, который можно указать, используя его имя, адрес HTTP или адрес IP. При выборе этой команды появляется диалоговое окно **Add Server**, приведенное на рис. П.82.

Рис. П.82. Диалоговое окно **Add Server**

Команда Tools, Customize Toolbox

Данная команда меню позволяет настроить вид панели **Toolbox**. При ее выборе появляется диалоговое окно **Customize Toolbox**, изображенное на рис. П.83.

Рис. П.83. Диалоговое окно **Customize Toolbox**

Раскрытая на рис. П.83 панель COM Controls позволяет добавить в список ресурсов панели **Toolbox** элементы управления COM.

Команда Tools, Add-in Manager

Данная команда меню позволяет добавить в среду программирования Visual Studio.NET дополнительные компоненты. При ее выборе появляется диалоговое окно **Add-in Manager**, изображенное на рис. П.84.

Каждая строка в окне списка данного диалогового окна содержит имя подключенного компонента и определение того, будет ли он подключаться при запуске среды программирования и будет ли он доступен при запуске компилятора из командной строки. Кроме того, данное диалоговое окно содержит текстовое поле, в которое будет выводиться краткое описание выделенного компонента, составленное его разработчиком.

Команда Tools, Build Comment Web Pages

При выборе данной команды меню появляется диалоговое окно **Build Comment Web Pages**, изображенное на рис. П.85.

Рис. П.84. Диалоговое окно **Add-in Manager**

Рис. П.85. Диалоговое окно **Build Comment Web Pages**

Это диалоговое окно позволяет создать набор страниц в формате HTML, содержащих описание всего решения или некоторых составляющих его проектов. Пример такого набора приведен на рис. П.86.

Рис. П.86. Вывод информации о приложении

Команда Tools, Macros

Данная команда меню не является терминальной. При ее выборе появляется контекстное меню для работы с макросами, изображенное на рис. П.87.

Данное контекстное меню позволяет записать или воспроизвести простой макрос, или редактировать записанную последовательность нажатия клавиш, добавляя в нее операторы VBScript.

Команда Tools, Trace Tool

Выбор данной команды меню запускает приложение трассировщика, для настройки параметров которого используется диалоговое окно **Dynamic ATL Trace Settings**, изображенное на рис. П.88.

Рис. П.87. Контекстное меню **Macros**

Рис. П.88. Диалоговое окно **Dynamic ATL Trace Settings**

Команда Tools, OLE/COM Object Viewer

Выбор данной команды меню запускает приложение **OLE/COM Object Viewer** , окно которого изображено на рис. П.89.

Рис. П.89. Окно приложения **OLE/COM Object Viewer**

Это приложение позволяет получить подробную информацию по объектам COM, установленным на данном компьютере.

Команда Tools, Spy+

Выбор данной команды меню запускает приложение **Spy+**, окно которого изображено на рис. П.90.

Это приложение позволяет контролировать совместную работу приложений на данном компьютере.

Команда Tools, External Tools

При выборе данной команды меню появляется диалоговое окно **External Tools**, изображеное на рис. П.91.

Это диалоговое окно используется для добавления и исключения из меню Tools команд вызова внешних приложений, а также для настройки параметров их вызова. Список имеющихся команд для вызова этих приложений выводится в окне списка **Menu Contents:**.

Рис. П.90. Окно приложения **Spy+**

Рис. П.91. Окно приложения **External Tools**

Команда Tools, Customize

Выбор данной команды меню выводит на экран диалоговое окно **Customize**, изображенное на рис. П.92.

Рис. П.92. Диалоговое окно **Customize**

Данное диалоговое окно позволяет произвести настройку внешнего вида и других параметров среды программирования Visual Studio.NET. Вкладка **Toolbars** данного диалогового окна позволяет определить набор выводимых на экран панелей инструментов. Вкладка **Commands** позволяет добавить или удалить кнопки в панели инструментов. Вкладка **Options** позволяет настроить режим вывода элементов управления среды программирования: задать размер кнопок панелей инструментов (крупные или нормальные) и режим вывода подсказок к ним, задать прозрачность плавающих панелей инструментов и произвести другие действия. Нажатие кнопки **Keyboard...** выводит диалоговое окно **Options**, позволяющее определить режим использования клавиатуры. Среди прочего, оно дает возможность задать режим использования функциональных клавиш в соответствии с режимами, используемыми в некоторых популярных средах программирования, и определить комбинации клавиш для выбранных команд.

Команда Tools, Options

Выбор данной команды меню выводит на экран диалоговое окно **Options**, изображенное на рис. П.93.

Данное диалоговое окно позволяет настраивать различные параметры среды программирования Visual Studio.NET.

Рис. П.93. Диалоговое окно **Options**

Меню Window

Меню **Window**, приведенное на рис. П.94, содержит команды выбора окон и их автоматического упорядочения на экране.

Команда Window, New Window

Выбор этой команды меню создает новое окно редактирования, в которое помещается копия файла, которому соответствовало окно редактирования, имевшее фокус ввода в момент выбора данной команды меню. В строке заголовка к имени файла добавляется номер окна (например, :2). Если окно, имевшее до этого фокус ввода, не содержало в своем заголовке номер окна, к имени файла в его заголовке добавляется номер :1. Изменения, произведенные в одном окне, немедленно отображаются в других окнах.

Команда Window, Split

Выбор данной команды меню разделяет по горизонтали окно редактирования текстового файла, имеющее в данный момент фокус ввода, на две панели, как это показано на рис. П.95. В каждой из этих панелей отображается текст одного и того же файла, но каждая из них имеет независимые друг от друга вертикальную и горизонтальную полосы прокрутки, что позволяет одновременно просматривать различные фрагменты текста этого файла.

Рис. П.94. Меню **Window**

Рис. П.95. Разделенное окно

Чтобы убрать разделение окна выберите команду **Window, Remove Split**, появившуюся на месте команды **Window, Split**, или перетащите разделительную полосу между панелями в крайнюю верхнюю позицию.

Команда Window, Dockable

Данная команда меню управляет состоянием окон среды программирования Visual Studio.NET, таких как **Workspace**, **Output**, **Watch** и подобных им окон. Если такое окно имеет фокус ввода, и данная команда меню отмечена (у нее стоит флажок), то это окно может стать стационарным и отображаться как панель соответствующего окна Visual Studio.NET.

Команда Window, Hide

Данная команда меню управляет состоянием окон среды программирования Visual Studio.NET, таких как **Workspace**, **Output**, **Watch** и подобных им окон. При ее выборе окно, имеющее фокус ввода, скрывается (убирается с экрана). После вывода на экран соответствующего окна с использованием меню **View** оно появляется в том месте экрана и с теми установками, с которыми оно было скрыто.

Команда Window, Floating

Данная команда меню управляет состоянием окон среды программирования Visual Studio.NET, таких как **Workspace**, **Output**, **Watch** и подобных им окон. При ее выборе окно, имеющее фокус ввода, теряет возможность фиксации (превращения в панель). Если в момент выбора команды это окно было зафиксировано, и отображалось как панель другого окна, оно становится плавающим. Если оно уже было плавающим, то оно не претерпевает никаких внешних изменений, но попытки превратить его в панель завершаются безуспешно.

Команда Window, Auto Hide

Данная команда меню управляет состоянием окон среды программирования Visual Studio.NET, таких как **Workspace**, **Output**, **Watch** и подобных им окон. При ее выборе окно с вкладками, имеющее фокус ввода, само превращается в панель окна Visual Studio.NET, а ее вкладки становятся панелями, как это показано на рис. П.96.

Для обратного преобразования нужно сделать активной одну из вкладок бывшего окна и снова выбрать эту команду меню (снять с нее флажок).

Рис. П.96. Превращение вкладок в панели

Команда Window, Auto Hide All

Данная команда меню управляет состоянием окон среды программирования Visual Studio.NET, таких как **Workspace**, **Output**, **Watch** и подобных им окон. При ее выборе все эти окна с вкладками превращаются в панели окна Visual Studio.NET как это показано на рис. П.97.

Обратное преобразование каждого окна производится отдельно, как это было описано в предыдущем разделе.

Команда Window, New Horizontal Tab Group

Данная команда меню управляет состоянием окон редактирования среды программирования Visual Studio.NET. При ее выборе активное окно редактирования, содержащее несколько вкладок, разбивается по горизонтали на два независимых окна, и в нижнее окно помещается его текущая вкладка, которая при этом исчезает из верхнего окна, как это показано на рис. П. 98.

Вкладки можно перетаскивать из одного окна в другое. После того как из окна будет перетащена последняя вкладка, окно уничтожается, а на его место автоматически растягивается соседнее окно.

Рис. П.97. Превращение всех окон в панели

Рис. П.98. Создание нового горизонтального окна с вкладками

После создания горизонтального окна с вкладками в меню исчезает команда, позволяющая создать вертикальное окно с вкладками.

Команда Window, New Vertical Tab Group

Данная команда меню управляет состоянием окон редактирования среды программирования Visual Studio.NET. При ее выборе активное окно редактирования, содержащее несколько вкладок, разбивается по вертикали на два независимых окна, и в правое окно помещается его текущая вкладка, которая при этом исчезает из левого окна, как это показано на рис. П.99.

Рис. П.99. Создание нового вертикального окна с вкладками

Вкладки можно перетаскивать из одного окна в другое. После того, как из окна будет перетащена последняя вкладка, окно уничтожается, а на его место автоматически растягивается соседнее окно.

После создания вертикального окна с вкладками в меню исчезает команда, позволяющая создать горизонтальное окно с вкладками.

Команда Window, Move to Next Tab Group

Данная команда меню управляет состоянием окон редактирования среды программирования Visual Studio.NET. При ее выборе активная вкладка перемещается в следующее окно. Если это окно является последним, данная команда в меню отсутствует.

Команда Window, Move to Previous Tab Group

Данная команда меню управляет состоянием окон редактирования среды программирования Visual Studio.NET. При ее выборе активная вкладка перемещается в предыдущее окно. Если это окно является первым, данная команда в меню отсутствует.

Команда Window, Close All Documents

Данная команда меню управляет состоянием окон редактирования среды программирования Visual Studio.NET. Ее выбор приводит к закрытию всех окон документов в приложении.

Список открытых окон

Нижняя часть меню **Window** содержит список открытых окон, что позволяет достаточно просто перемещаться между ними. Если количество открытых окон больше десяти, то в этом списке отображаются только десять окон, расположенные в порядке убывания времени последнего обращения к ним пользователя. Для доступа к остальным раскрытым окнам следует выбрать команду меню **Window**, **Windows…**.

Примечание

Ценность списка открытых окон меню **Windows** существенно снижена превращением окон редактирования во вкладки, что позволяет открывать их по ярлыку. Этот список можно рассматривать как рудимент предыдущих версий.

Команда Window, Windows

Выбор данной команды меню выводит на экран диалоговое окно **Windows**, изображенное на рис. П.100.

Данное диалоговое окно позволяет предавать фокус ввода окну, выделенному в окне списка, а также сохранять содержимое окон, выделенных в данном окне списка, и закрывать эти окна.

Рис. П.100. Диалоговое окно **Windows**

Меню Help

Меню **Help**, приведенное на рис. П.101, содержит команды обращения к справочной системе Visual Studio.NET. Как уже говорилось выше, эта справочная система может выводиться в окне среды программирования Visual Studio.NET или в окнах отдельного приложения справочной системы. В данном случае нами рассматривается установленный по умолчанию режим вывода в окна среды Visual Studio.NET.

Команда Help, Dynamic Help (<Ctrl>+<F1>)

Данная команда меню используется для получения краткой справки по элементу среды программирования, с которым в данный момент производятся некоторые действия. Информация выводится в окно **Dynamic Help**, изображенное на рис. П.102.

Команда Help, Contents (<Ctrl>+<Alt>+<F1>)

При выборе данной команды меню на экран выводится окно **Contents**, изображенное на рис. П.103. Если это окно уже выведено на экран, ему просто передается фокус ввода.

Данное окно позволяет получить доступ к интерактивной справке системы программирования Visual Studio.NET.

Команда Help, Index (<Ctrl>+<Alt>+<F2>)

При выборе данной команды меню на экран выводится окно **Index**, изображенное на рис. П.104. Если это окно уже выведено на экран, ему просто передается фокус ввода.

Рис. П.101. Меню **Help**

Рис. П.102. Окно **Dynamic Help**

Рис. П.103. Окно **Contents**

Рис. П.104. Окно **Index**

Данное окно позволяет произвести поиск по ключу в интерактивной справке системы программирования Visual Studio.NET.

Команда Help, Search (<Ctrl>+<Alt>+<F3>)

При выборе данной команды меню на экран выводится окно **Search**, изображенное на рис. П.105. Если это окно уже выведено на экран, ему просто передается фокус ввода.

Данное окно позволяет получить произвести поиск текста в интерактивной справке системы программирования Visual Studio.NET.

Команда Help, Index results (<Shift>+<Alt>+<F2>)

Если результаты поиска, проведенного по ключу, заданному в окне **Index**, не позволили однозначно определить выводимую тему справки, среда программирования Visual Studio.NET выводит окно **Index Results**, изображенное на рис. П.106, содержащее темы, из которых следует сделать выбор.

При выборе соответствующей темы справка по ней будет выведена в новую вкладку окна редактирования.

Рис. П.105. Окно **Search**

Рис. П.106. Окно **Index Results**

Команда меню **Help**, **Index results...** используется для передачи фокуса ввода этому окну из любого другого окна среды программирования.

Команда Help, Search results (<Shift>+<Alt>+<F3>)

Результаты поиска текста, заданного в окне **Search** выводятся в окно **Search Results**, изображенное на рис. П.107.

При выборе соответствующей темы справка по ней будет выведена в новую вкладку окна редактирования.

Команда меню **Help**, **Search results...** используется для передачи фокуса ввода этому окну из любого другого окна среды программирования.

Команда Help, Edit Filters

При выборе данной команды меню на экран выводится окно **Edit Filters**, изображенное на рис. П.108.

Данное окно позволяет задать фильтры, используемые при поиске информации в справочной системе среды Visual Studio.NET. Задание фильтров позволяет отсеять информацию, не относящуюся к поставленному вопросу за счет конкретизации области поиска.

Рис. П.107. Окно **Search Results**

Рис. П.108. Окно **Edit Filters**

Остальные команды меню

Остальные команды меню используются для связи с центрами технической поддержки Microsoft и получения от них необходимой информации и новых версий продукта. Поскольку объем этой программы существенно сокращен и не удовлетворяет минимальным потребностям рядового пользователя, эти команды описываться не будут.

Окна Visual Studio.NET

При описании команд меню часто упоминались различные стандартные окна среды программирования Visual Studio.NET. Рассмотрим подробнее некоторые из них.

Окно Solution Explorer

Окно **Solution Explorer**, изображенное на рис. П.109, используется для работы с файлами, включенными в проект. В C# это окно имеет достаточно простую структуру, поскольку в проекты этого языка не включаются файлы ресурсов и файлы заголовков, как это имеет место в проектах C++. Поэтому файлы одного проекта не разделяются по отдельным папкам, а содержатся непосредственно в папке проекта. Двойной щелчок левой кнопкой мыши по имени файла приводит к открытию окна редактирования данного файла. Нажатие клавиши **<Delete>** приводит к удалению выделенного файла из проекта.

Окно Class View

Окно **Class View**, изображенное на рис. П.110, используется для автоматизации работы с классами приложения.

Верхним уровнем иерархии в данном окне являются папки проекта, содержащие папки составляющих его именованных областей и типов. Каждая папка именованной области или типа, в свою очередь, содержит папки включенных в нее именованных областей и типов, а также значки членов данного типа, если это папка какого-либо типа. Защищенные члены класса (`protected`) отмечаются ключом, а закрытые члены класса (`private`) — замком.

Примечание

Для удобства читателя на рис. П.109 и рис. П.110 повторно приведены окна **Solution Explorer и Class View** соответственно, ранее изображенные на рис. П.47 и рис. П.48.

При щелчке правой кнопкой мыши по имени типа проекта появляется контекстное меню, изображенное на рис. П.111.

Рис. П.109. Окно **Solution Explorer**

Рис. П.110. Окно **Class View**

Рис. П.111. Окно **Class View** с раскрытым контекстным меню

Рассмотрим наиболее употребительные команды данного меню:

- **Go to Definition** (Перейти к описанию) — открывает файл, содержащий описание данного типа и помещает текстовый курсор в первой строке описания.
- **Add** (Добавить) — данная команда не является терминальной. При ее вызове появляется контекстное меню, содержащее следующие команды (если выделен класс):
  - **Add Method...** (Добавить метод) — дублирует одноименную команду меню **Property**.
  - **Add Property...** (Добавить свойство) — дублирует одноименную команду меню **Property**.
  - **Add Field...** (Добавить поле) — дублирует одноименную команду меню **Property**.
  - **Add Indexer...** (Добавить индексатор) — дублирует одноименную команду меню **Property**.
- **Copy** (Копировать) — копирует в буфер обмена полное имя выделенного компонента.
- **Sort Alphabetically** (Упорядочит по алфавиту) — упорядочивает элементы в папках и сами папки в алфавитном порядке.
- **Sort by Type** (Упорядочить по типу) — упорядочивает элементы в папках по типу.
- **Sort by Access** (Упорядочить по режиму доступа) — упорядочивает элементы в папках по мере ужесточения режима доступа к ним.
- **Group by Type** (Группировать по типу) — группирует элементы по типу (классы, интерфейсы, методы, поля, индексаторы и т. д.).
- **Properties** (Свойства) — выводит окно Properties, которое будет рассмотрено ниже.

Окно Properties

Окно **Properties**, изображенное на рис. П.112, используется для вывода информации о различных компонентах программы. Основным назначением этого окна является работа с ресурсами приложения. Поскольку в C# ресурсы не используются, это окно, как правило, не используется.

Окно Watch

Окно **Watch**, изображенное на рис. П.113, появляется в процессе выполнения программы и служит для получения информации о значениях переменных, выбранных пользователем. Если тип переменной, содержащейся в данном окне, является ссылкой на объект, то эта переменная выводится в виде папки, позволяющей получить доступ к полям данного объекта.

Рис. П.112. Окно **Properties**

Рис. П.113. Окно **Watch**

Помимо окна **Watch** при запуске приложения на исполнение также выводятся окна **Autos** и **Locals**, имеющие одинаковую с ним структуру полей. В первом из них выводятся значения переменных, выбираемых средой программирования, а во втором — значения локальных переменных.

Окно Breakpoints

Окно **Breakpoints**, изображенное на рис. П.114, появляется в процессе выполнения программы и используется для работы с точками останова.

Для изменения свойств точки останова, указанных в этом окне, достаточно щелкнуть на нем правой кнопкой мыши и выбрать в появившемся контекстном меню команду **Properties**. Появится диалоговое окно **Breakpoint Properties**, изображенное на рис. П.115. В этом окне будут отражены все текущие установки для данной точки останова.

Рис. П.114. Окно **Breakpoints**

Рис. П.115. Окно **Breakpoint Properties**

Предметный указатель

C, I

Common Language Runtime 26
Intermediate Language 26

А

Автоматическое управление памятью 53
Атрибут 369, 372
  AttributeUsage 378
  comimport 384
  comvisibility 386
  conditional 380, 439
  dispid 387
  DllImport 320, 388
  globalobject 390
  guid 390
  hasdefaultinterface 391
  importedfromcom 392
  in 393
  interfacetype 393
  iscomregisterfunction 394
  marshal 395
  name 397
  noidispatch 398
  nonserialized 407, 432
  obsolete 408
  out 398
  predeclared 399
  returnshresult 399
  serializable 407, 432
  structlayout 401
  structoffset 403
  typelibfunc 404
  typelibtype 405
  typelibvar 406

Б

Библиотека динамической компоновки 25
Блок 69
Бордюр 479

В

Виртуальные функции 25
Виртуальный метод 315
Вкладка
  Server Explorer 515
  Toolbox 516
Выражение 195
  арифметическое 243
    операция - 248
    операция % 245
    операция * 243
    операция / 244
    операция + 246
  значение 196
  константное 264
  логические операции 258
  логическое 265
  отношения 250
  первичное 215
    base 227
    this 227
    в скобках 219
    вызов делегата 224
    вызов метода 223
    вызов функционального члена 222
    доступ к символу строки 226
    доступ к члену 219
    доступ к элементу 224
    доступ к элементу массива 225
    доступ по индексатору 225
    литерал 215
    операция checked 237
    операция new 229
    операция sizeof 237
    операция typeof 236
    операция unchecked 237
    постфиксного декрементирования 228
    постфиксного инкрементирования 228
    простое имя 216

создание делегата 233
создание массива 231
создание объекта 230
побитовые операции 256
присваивания 261
сдвига 249
унарное
операция - 239
операция ! 240
операция ~ 240
операция + 239
преобразование типа 242
префиксное декрементирование 241
префиксное инкрементирование 241
условное 260

Д

Делегат
ThreadStart 434
Диалоговое окно
Add Existing Item 527
Add New Item 523
Add Reference 528
Add Server 541
Add Web Reference 529
Add-in Manager 542
Advanced Save Options 489
Batch Build 531
Breakpoint Properties 566
Build Comment Web Pages 542
C# Class Wizard 524
C# Field Wizard 526
C# Indexer Wizard 526
C# Method Wizard 524
C# Property Wizard 525
Configuration Manager 440, 531
Customize 547
Customize Toolbox 542
Data Link Properties 540
Dynamic ATL Trace Settings 544
External Tools 546
Find 495
Find in Files 498
Find Symbol 501
Go To 503
Insert File 504
New Breakpoint 538
New File 483
New Project 483
Open File 485
Open Project 485
Open Project From Web 485, 487
Open Solution 487
Open With 486
Options 548
Page Setup 490
Print 491
Processes 534
Property Pages 521
QuickWatch 538
Replace 497
Replace in Files 501
Save crashdump As... 538
Save File As 488
Save View 520
Selected Components 502
Windows 555
Директива using 47

З

Значение:
бесконечность 152
неопределенное 152
нулевое 152

И

Иерархия классов 24
Инкапсуляция 22
Инструкция 69
Интерфейс:
ICustomFormatter 428
IFormattable 427
IServiceObjectProvider 428
базовый 350
явный базовый 349
Исключение 99
System FormatException 430
System.OverflowException 452

К

Класс 369
BooleanSwitch 444
PrincipalPermission 471
Stream 431
Switch 447
System
Array 161, 286
Attribute 97, 369
Console 47
CriticalSection 295
Delegate 88, 165
Enum 156
Exception 99, 290
ICloneable 177
Object 159
String 159, 415
System.Diagnostics
Debug 440
Trace 440
TraceSwitch 445
Абстрактный 298
скрытие членов 302
функциональный член 301
Ключевое слово
base 319
class 297
const 304
delegate 166
enum 156
interface 349
struct 146
Кнопка панели инструментов
Add New Item 481
Class View 481
Copy 481
Cut 481
Find in Files 481
Navigate Backward 481
Navigate Forward 481
New Project 481
Open File 481
Paste 481
Properties Window 481
Redo 481
Save 481
Save All 481
Solution Explorer 481
Start 481
Toolbox 481
Undo 481
Коллекция
Debug
Listeners 442
TraceListeners 442
Команда
File Exit 492
Команда меню
Build
Batch Build 531
Build 530
Configuration Manager 440, 531
Deploy 531
Rebuild 530
Debug
Apply Code Changes 534
Break 534
Clear All Breakpoints 538
Detach All 534
Disable Breakpoint 540
Go To Disassembly 537
New Breakpoint 538
Processes 534
QuickWatch 538
Restart 534
Run to Cursor 537
Save As CrashDump 538
Set Next Statement 537
Start 533
Start Without Debugging 534
Step By 536
Step Into 535
Step Out 536
Step Over 535
Stop Debugging 534
Windows 533
Edit
Advanced 504
Bookmarks 506
Copy 494
Cut 494
Delete 494
Find and Replace 495
Insert File As Text 504
IntelliSense 510

(продолжение рубрики на стр. 570)

Outlining 507
Paste 494
Redo 493
Select All 494
Undo 492
Go To 503
File
Advanced Save Options 489
Close 487
Close Solution 488
New 483
Open 484
Open Solution 487
Page Setup 490
Print 490
Recent Files 491
Recent Projects 491
Save 488
Save All 490
Source Control 490
Add Project 487
Save As 488
Help
Contents 556
Dynamic Help 556
Edit Filters 561
Index 556
Index results 559
Search 559
Search results 561
Show Start Page 475
Project
Add Class 524
Add Component 524
Add Existing Item 527
Add Field 526
Add Indexer 526
Add Inherited Control 523
Add Inherited Form 523
Add Method 524
Add New Item 526
Add Property 525
Add Reference 528
Add User Control 523
Add Web Reference 529
Add Windows Form 523
Exclude From Project 528
Set as StartUp Project 529
Show All Files 528
Tools
Add-in Manager 542
Build Comment Web Pages 542
Connect to Database 540
Connect to Server 541
Customize Toolbox 542
External Tools 546
Macros 544
OLE/COM Object Viewer 546
Spy+ 546
Trace Tool 544
Customize 547
Options 548
View
Class View 515
Code 513
Navigate Backward 520
Navigate Forward 521
Open 513
Open With 513
Other Windows 517
Properties Window 515
Property Pages 521
Resource View 515
Server Explorer 515
Server Explorer View 520
Show Tasks 517
Solution Explorer 514
Toolbars 519
Toolbox 516
Web Browser 517
View Full Screen 519
Window
Auto Hide 551
Auto Hide All 552
Dockable 551
Floating 551
Hide 551
Move to Next Tab Group 555
Move to Previous Tab Group 555
New Horizontal Tab Group 552
New Vertical Tab Group 554
New Window 549
Split 549
Close All Documents 555
Windows 555

Комментарий 107
Компилятор времени выполнения приложения 28
Константное выражение 304

Л

Литерал 110
  логический 110, 215
  символьный 112, 216
  строковый 113, 216
  формата с плавающей точкой 111, 216
  целочисленный 110, 215

М

Манифест 44
Массив 160
Меню
  Build 529
  Debug 532
  Edit 492
  File 483
  Help 556
  Project 522
  Tools 540
  View 513
  Window 549
Метаданные 27
Метод
  Debug
    Assert 441
    Write 442
    WriteIf 442
    WriteLine 442
    WriteLineIf 442
  Format 417
  ICustomFormatter
    Format 428
  IFormattable
    Format 427
  IServiceObjectProvider
    GetServiceObject 428
  Main 47
  object
    ToString 63
  System
    Collect 55
    Console
      ReadLine 49
      Write 49
      WriteLine 48
    GC
      RequestFinalizeOnShutdown 346
      SupressFinalize 346
  Thread
    Sleep 434
    Start 434
    Stop 434
  Trace
    Assert 441
    Write 442
    WriteIf 442
    WriteLine 442
    WriteLineIf 442
  TraceListeners
    Add 442
    Clear 442
  абстрактный 319
  тело 320
  формальные параметры 311
Механизм
  execution engine 35
Модификатор:
  abstract 298, 319
  internal 302
  override 318
  private 302
  protected 302
  public 302
  readonly 306
  sealed 298
  static 303
  virtual 315

Н

Наследование 22, 301

О

Область видимости имен 301
Объект 373
  System
    Type 295

Объектно-ориентированное
программирование 21
Окно
Autos 566
Breakpoints 566
Class View 515, 563
Contents 556
Dynamic Help 556
Edit Filters 561
Index 556
Index Results 559
Locals 566
Properties 515, 565
Resource View 515
Search 559
Search Results 561
Solution Explorer 514, 562
Watch 565
Операнд 197
Оператор
break 70, 288
catch 456
checked 73, 294, 452
continue 289
do 71, 283
finally 463
for 72, 284
foreach 72, 285
goto 69, 289
if ... else 70
lock 74, 295, 435
return 73, 290
switch 70
throw 73, 290
try 73, 291, 456
unchecked 73, 294, 452
while 71, 282
блок 269
выражение 275
доступный 267
конечная точка 267
объявления 271
переключатель 277
перехода 287
помеченный 270
последовательность 269
пустой 270
составной 69, 269
условный 275
цикла 282
Операция 197
as 256, 366
is 256, 366
асоциативность 197
несовместимость типов операндов 204
перегрузка 200
поиск реализации 201
преобразование типов операндов 202
приоритет 197
сравнения 251

П

Панель инструментов:
Standard 480, 493
плавающая 479
фиксированная 479
Параметры 371
Перегрузка метода 315
Переменная 63
локальная 64
Переполнение 452
Перечислимый тип
AttributeTargets 410
CallingConvention 411
CharSet 411, 412
COMVisibility 412
LayoutKind 412
UnmanagedType 412
Полиморфизм 22, 315
Преобразование типов 339
неявное 62
явное 62
Преобразование типов 339
Препроцессор
директива
#define 102
#elif 104
#else 104
#endif 104
#error 104
#if 104
#line 105
#undef 102
#warning 104

Приложение
OLE/COM Object Viewer 546
Spy+ 546
Пространство имен 91
System.Diagnostics 440
глобальное 115

Р

Распаковка значения 63
Режимы доступа 119
Рефлексия 33

С

Сборка 44, 91
Свойство:
для чтения и записи 323
процедура доступа 77, 322
статическое 322
только для записи 323
только для чтения 323
Сдвиг
арифметический 250
логический 250
Сигнатура 125
Система
Virtual Execution System 35
виртуальных объектов 30
Скрытие данных 23
Спецификация
Common Language Specification 33
Структура
System
Int16 149
Int32 149
Int64 144

Т

Технология
COM 26
Тип:
делегата 330
доступ к члену 205
значения 143
десятичный 154
инициализация 145
логический 155
перечислимый 90, 155
простой 148
распаковка 184
с плавающей точкой 152
структура 86, 146
упаковка 177
целочисленнный 150
Ссылки 159
делегат 88, 165
интерфейс 87
класс 74, 159
Функциональные члены 206
вызов 213
вызов как метода типа object 214
поиск реализации 211
список аргументов 209

У

Упаковка значений 63

Ф

Формат
Portable Executable 27

Ч

Член:
доступный 119
недоступный 119
область видимости 127
область действия 120
интерфейса 350
доступ 352
полное имя 354
явная реализация 355
класса
деструктор 83, 345
индексатор 81, 335
константа 74, 304
конструктор объекта 82, 341
метод 76, 310
нестатический 302
операция 80, 338
поле 75, 306
свойство 77, 321
событие 78, 330
статический 302
статический конструктор 83, 347