

Содержание

Об авторах	33
О технических редакторах	34
Благодарности	34
Введение	35
Важность .NET и C#	35
Преимущества .NET	36
Что нового в .NET Framework 4.5	38
Асинхронное программирование	38
Приложения Windows Store и среда Windows Runtime	38
Усовершенствования доступа к данным	38
Усовершенствования WPF	38
Инфраструктура ASP.NET MVC	39
Для чего подходит C#	39
Что необходимо для написания и выполнения кода C#	41
Как организована эта книга	41
Часть I. Язык C#	41
Часть II. Visual Studio	41
Часть III. Основы	41
Часть IV. Данные	42
Часть V. Презентация	42
Часть VI. Коммуникации	42
Соглашения	42
Исходный код	43
От издательства	43
Часть I. Язык C#	44
Глава 1. Архитектура .NET	45
Отношение между C# и .NET	46
Общезыковая исполняющая среда	46
Независимость от платформы	47
Повышение производительности	47
Способность к взаимодействию на уровне языка	47
Visual C++ 2012	48
Более внимательный взгляд на промежуточный язык (IL)	49
Поддержка объектной ориентации и интерфейсов	50
Различие типов значений и ссылочных типов	51
Строгая типизация данных	51
Обработка ошибок с помощью исключений	57
Применение атрибутов	58
Сборки	58
Закрытые сборки	59
Разделяемые сборки	59
Рефлексия	60
Параллельное программирование	60
Асинхронное программирование	60

Классы .NET Framework	61
Пространства имен	62
Создание приложений .NET с использованием C#	62
Создание приложений ASP.NET	63
Веб-элементы управления	64
ASP.NET MVC	64
ASP.NET Dynamic Data	64
ASP.NET Web API	64
Windows Presentation Foundation (WPF)	65
Приложения Windows 8	65
Службы Windows	65
Windows Communication Foundation (WCF)	66
Windows Workflow Foundation (WF)	66
Роль языка C# в архитектуре .NET для корпоративных приложений	66
Резюме	68
Глава 2. Основы C#	69
Основы C#	70
Первая программа на C#	70
Код	70
Компиляция и запуск программы	70
Более пристальный взгляд на код	71
Переменные	73
Инициализация переменных	73
Выведение типа	74
Область видимости переменных	75
Константы	78
Предопределенные типы данных	78
Типы значений и ссылочные типы	78
Типы CTS	80
Предопределенные типы значений	80
Предопределенные ссылочные типы	83
Управление потоком выполнения	85
Условные операторы	85
Циклы	88
Операторы перехода	91
Перечисления	92
Пространства имен	94
Директива using	95
Псевдонимы пространств имен	96
Метод Main()	97
Несколько методов Main()	97
Передача аргументов в Main()	98
Дополнительные сведения о компиляции файлов C#	99
Консольный ввод-вывод	100
Использование комментариев	102
Внутренние комментарии в исходных файлах	103
Документация XML	103
Директивы препроцессора C#	105

#define и #undef	106
#if, #elif, #else и #endif	106
#warning и #error	107
#region и #endregion	107
#line	108
#pragma	108
Рекомендации по программированию на C#	108
Правила для идентификаторов	108
Соглашения по использованию	110
Резюме	115
Глава 3. Объекты и типы	116
Создание и использование классов	117
Классы и структуры	117
Классы	118
Данные-члены	118
Функции-члены	118
Анонимные типы	132
Структуры	132
Структуры являются типами значений	134
Структуры и наследование	135
Конструкторы для структур	135
Слабые ссылки	135
Частичные классы	136
Статические классы	138
Класс Object	138
Методы System.Object	138
Метод ToString()	139
Расширяющие методы	141
Резюме	141
Глава 4. Наследование	142
Концепция наследования	143
Типы наследования	143
Сравнение наследования реализации и наследования интерфейса	143
Множественное наследование	143
Структуры и классы	144
Наследование реализации	144
Виртуальные методы	145
Соккрытие методов	146
Вызов базовых версий функций	147
Абстрактные классы и функции	147
Запечатанные классы и методы	148
Конструкторы производных классов	149
Добавление в иерархию конструктора	150
Добавление в иерархию конструкторов с параметрами	152
Модификаторы	153
Модификаторы видимости	153
Другие модификаторы	154

Интерфейсы	155
Определение и реализация интерфейсов	156
Производные интерфейсы	159
Резюме	160
Глава 5. Обобщения	161
Обзор обобщений	162
Производительность	162
Безопасность типов	163
Повторное использование двоичного кода	164
“Разбухание” кода	164
Рекомендации по именованию	164
Создание обобщенных классов	165
Возможности обобщений	168
Стандартные значения	169
Ограничения	169
Наследование	171
Статические члены	172
Обобщенные интерфейсы	172
Ковариантность и контравариантность	173
Ковариантность обобщенных интерфейсов	174
Контравариантность обобщенных интерфейсов	175
Обобщенные структуры	176
Обобщенные методы	178
Пример обобщенного метода	179
Обобщенные методы с ограничениями	179
Обобщенные методы с делегатами	180
Специализация обобщенных методов	181
Резюме	182
Глава 6. Массивы и кортежи	183
Множество объектов одного и того же и разных типов	184
Простые массивы	184
Объявление массива	184
Инициализация массива	184
Доступ к элементам массива	185
Использование ссылочных типов	186
Многомерные массивы	187
Зубчатые массивы	188
Класс <code>Array</code>	189
Создание массивов	189
Копирование массивов	190
Сортировка	191
Использование массивов в качестве параметров	194
Ковариантность массивов	194
Структура <code>ArraySegment<T></code>	194
Перечисления	195
Интерфейс <code>IEnumerator</code>	196
Оператор <code>foreach</code>	196
Оператор <code>yield</code>	196

Кортежи	201
Структурное сравнение	202
Резюме	204
Глава 7. Операции и приведения	205
Понятие операций и приведений	206
Операции	206
Сокращенные версии операций	207
Условная операция	208
Операции <code>checked</code> и <code>unchecked</code>	209
Операция <code>is</code>	210
Операция <code>as</code>	210
Операция <code>sizeof</code>	210
Операция <code>typeof</code>	211
Типы, допускающие <code>null</code> , и операции	211
Операция поглощения <code>null</code>	211
Приоритеты операций	212
Безопасность типов	212
Преобразования типов	213
Неявные преобразования	213
Явные преобразования	214
Упаковка и распаковка	216
Проверка объектов на предмет равенства	217
Проверка ссылочных типов на равенство	217
Проверка типов значений на равенство	218
Перегрузка операций	219
Функционирование операций	220
Пример перегрузки операции: структура <code>Vector</code>	221
Операции, допускающие перегрузку	228
Пользовательские приведения	228
Реализация пользовательских приведений	229
Множественное приведение	236
Резюме	239
Глава 8. Делегаты, лямбда-выражения и события	240
Ссылка на методы	241
Делегаты	241
Объявление делегатов	242
Использование делегатов	243
Пример простого делегата	246
Делегаты <code>Action<T></code> и <code>Func<T></code>	247
Пример <code>BubbleSorter</code>	248
Групповые делегаты	251
Анонимные методы	254
Лямбда-выражения	255
Параметры	255
Множество строк кода	256
Замыкания	256
Замыкания и операторы <code>foreach</code>	257

События	258
Издатель события	258
Прослушиватель событий	260
Слабые события	261
Резюме	264
Глава 9. Строки и регулярные выражения	265
Исследование <code>System.String</code>	266
Построение строк	267
Члены класса <code>StringBuilder</code>	270
Форматирующие строки	271
Регулярные выражения	277
Введение в регулярные выражения	277
Пример <code>RegularExpressionsPlayaround</code>	279
Отображение результатов	281
Соответствия, группы и захваты	283
Резюме	285
Глава 10. Коллекции	286
Обзор	287
Интерфейсы и типы коллекций	287
Списки	288
Создание списков	290
Инициализаторы коллекций	290
Добавление элементов	291
Вставка элементов	291
Доступ к элементам	292
Удаление элементов	293
Поиск	294
Сортировка	295
Преобразование типов	297
Коллекции, доступные только для чтения	298
Очереди	298
Стеки	302
Связные списки	303
Сортированные списки	308
Словари	310
Тип ключа	310
Пример словаря	312
Списки поиска	316
Сортированные словари	316
Множества	317
Наблюдаемые коллекции	319
Битовые массивы	320
Класс <code>BitArray</code>	320
Структура <code>BitVector32</code>	322
Параллельные коллекции	325
Создание конвейеров	326
Использование класса <code>BlockingCollection<T></code>	328

Использование класса <code>ConcurrentDictionary<TKey, TValue></code>	330
Завершение построения конвейера	331
Производительность	333
Резюме	335
Глава 11. Язык интегрированных запросов	336
Обзор LINQ	337
Списки и сущности	337
Запрос LINQ	340
Расширяющие методы	341
Отложенное выполнение запросов	342
Стандартные операции запросов	344
Фильтрация	346
Фильтрация с индексом	346
Фильтрация на основе типа	347
Составная конструкция <code>from</code>	347
Сортировка	348
Группирование	349
Группирование с вложенными объектами	350
Внутреннее соединение	351
Левое внешнее соединение	352
Групповое соединение	353
Операции над множествами	356
Метод <code>Zip</code>	357
Разделение	358
Агрегатные операции	359
Операции преобразования	360
Генерирующие операции	362
Parallel LINQ	362
Параллельные запросы	362
Разделители	363
Отмена	364
Деревья выражений	364
Поставщики LINQ	367
Резюме	368
Глава 12. Динамические расширения языка	369
Динамическое программирование	370
Среда <code>Dynamic Language Runtime</code>	370
Тип <code>dynamic</code>	370
Особенности типа <code>dynamic</code>	371
Хостинг DLR с помощью объекта <code>ScriptRuntime</code>	374
<code>DynamicObject</code> и <code>ExpandableObject</code>	377
<code>DynamicObject</code>	377
<code>ExpandableObject</code>	379
Резюме	380
Глава 13. Асинхронное программирование	381
Важность асинхронного программирования	382

Асинхронные шаблоны	382
Синхронный вызов	389
Асинхронный шаблон	390
Асинхронный шаблон, основанный на событиях	391
Асинхронный шаблон, основанный на задачах	392
Основы асинхронного программирования	394
Создание задач	394
Вызов асинхронного метода	395
Задачи продолжения	395
Контекст синхронизации	396
Использование множества асинхронных методов	396
Преобразование асинхронного шаблона	397
Обработка ошибок	398
Обработка исключений, возникающих в асинхронных методах	399
Исключения, возникающие в нескольких асинхронных методах	399
Использование информации типа <code>AggregateException</code>	400
Отмена	401
Начало отмены	401
Отмена выполнения средств платформы	401
Отмена выполнения специальных задач	402
Резюме	402
Глава 14. Управление памятью и указатели	403
Управление памятью	404
Внутренние механизмы управления памятью	404
Типы значений	404
Ссылочные типы	406
Сборка мусора	408
Освобождение неуправляемых ресурсов	410
Деструкторы	410
Интерфейс <code>IDisposable</code>	411
Реализация интерфейса <code>IDisposable</code> и деструктора	413
Небезопасный код	414
Прямой доступ в память через указатели	414
Пример использования указателей: <code>PointerPlayaround</code>	423
Использование указателей для оптимизации производительности	427
Резюме	431
Глава 15. Рефлексия	432
Манипулирование и просмотр кода во время выполнения	433
Специальные атрибуты	433
Написание специальных атрибутов	434
Пример специального атрибута: <code>WhatsNewAttributes</code>	437
Использование рефлексии	440
Класс <code>System.Type</code>	440
Пример <code>TypeView</code>	443
Класс <code>Assembly</code>	445
Завершение примера <code>WhatsNewAttributes</code>	446
Резюме	449

Глава 16. Ошибки и исключения	450
Введение	451
Классы исключений	451
Перехват исключений	454
Реализация множества блоков catch	456
Перехват исключений из другого кода	460
Свойства класса System.Exception	460
Что происходит с необработанными исключениями?	461
Вложенные блоки try	461
Классы исключений, определяемые пользователем	463
Перехват исключений, определяемых пользователем	464
Генерация исключений, определяемых пользователем	466
Объявление классов исключений, определяемых пользователем	469
Информация о вызывающем компоненте	470
Резюме	472
Часть II. Visual Studio	473
Глава 17. Visual Studio 2012	474
Работа с Visual Studio 2012	475
Изменения в файле проекта	477
Выпуски Visual Studio	478
Настройки Visual Studio	478
Создание проекта	479
Поддержка нескольких целевых версий .NET Framework	480
Выбор типа проекта	481
Исследование и написание кода проекта	485
Окно Solution Explorer	485
Работа с редактором кода	490
Ознакомление с другими окнами	492
Упорядочение окон	496
Построение проекта	496
Построение, компиляция и компоновка проекта	496
Отладочная и окончательная сборки	497
Выбор конфигурации	499
Редактирование конфигураций	499
Отладка кода	501
Добавление точек останова	501
Использование подсказок о состоянии данных и визуализаторов отладчика	501
Мониторинг и изменение переменных	503
Исключения	504
Многопоточность	505
Средство IntelliTrace	506
Инструменты для рефакторинга	506
Архитектурные инструменты	507
Граф зависимостей	508
Диаграмма уровней	509

Анализ приложений	510
Диаграмма последовательности	510
Профилировщик	511
Визуализатор параллелизма	513
Анализ кода	513
Метрики кода	515
Модульные тесты	515
Создание модульных тестов	515
Запуск модульных тестов	516
Ожидаемые исключения	517
Тестирование всех путей в коде	518
Внешние зависимости	518
Инфраструктура Fakes Framework	521
Приложения Windows 8, WCF и WF	522
Создание приложений WCF в Visual Studio 2012	522
Создание приложений WF в Visual Studio 2012	523
Создание приложений Windows Store в Visual Studio 2012	524
Резюме	525
Глава 18. Развертывание	526
Развертывание как часть жизненного цикла приложения	527
Планирование развертывания	527
Обзор вариантов развертывания	527
Требования к развертыванию	528
Развертывание исполняющей среды .NET	529
Традиционное развертывание	529
Развертывание с помощью xсору	530
Развертывание веб-приложений с помощью xсору	530
Программа установки Windows	530
Технология ClickOnce	531
Как работает ClickOnce	531
Публикация приложения ClickOnce	532
Параметры ClickOnce	532
Кеш приложений для файлов ClickOnce	535
Установка приложения	536
API-интерфейс развертывания ClickOnce	536
Развертывание веб-приложений	537
Веб-приложение	537
Файлы конфигурации	538
Создание пакета Web Deploy	538
Приложения Windows 8	539
Создание пакета приложения	540
Комплект сертификации приложений Windows	541
Заливка	542
API-интерфейс развертывания Windows	543
Резюме	545

Часть III. Основы	547
Глава 19. Сборки	548
Что собой представляют сборки	549
Характеристики сборок	549
Структура сборки	550
Манифест сборки	551
Пространства имен, сборки и компоненты	551
Закрытые и разделяемые сборки	552
Подчиненные сборки	552
Просмотр содержимого сборок	552
Создание сборок	553
Создание модулей и сборок	553
Атрибуты сборок	554
Создание и загрузка сборок динамическим образом	556
Домены приложений	559
Разделяемые сборки	562
Строгие имена	562
Обеспечение целостности с использованием строгих имен	563
Глобальный кеш сборок	564
Создание разделяемой сборки	565
Создание строгого имени	565
Установка разделяемой сборки	566
Использование разделяемой сборки	566
Отложенное подписание сборок	567
Ссылки	568
Генератор машинных образов	569
Конфигурирование приложений .NET	570
Категории конфигурационных параметров	571
Привязка к сборкам	572
Контроль версий	573
Номера версий	574
Получение номера версии программным путем	574
Привязка к другим версиям сборки	575
Файлы политик издателя	576
Версия исполняющей среды	577
Разделение сборок между разными технологиями	578
Разделение исходного кода	578
Переносимая библиотека классов	579
Резюме	580
Глава 20. Диагностика	581
Обзор диагностики	582
Контракты кода	582
Предусловия	584
Постусловия	584
Инварианты	585
Чистота	586
Контракты для интерфейсов	586

Сокращения	588
Контракты и унаследованный код	588
Трассировка	588
Источники трассировки	590
Переключатели трассировки	591
Прослушиватели трассировки	592
Фильтры	594
Корреляция	595
Трассировка с помощью ETW	597
Протоколирование событий	599
Архитектура протоколирования событий	599
Классы протоколирования событий	601
Создание источника событий	602
Запись в журналы событий	603
Ресурсные файлы	603
Мониторинг производительности	607
Классы мониторинга производительности	607
Построитель счетчиков производительности	608
Добавление компонентов PerformanceCounter	610
Использование perfmon.exe	612
Резюме	613
Глава 21. Задачи, потоки и синхронизация	614
Обзор многопоточности	615
Класс Parallel	616
Организация циклов с помощью метода Parallel.For()	616
Организация циклов с помощью метода Parallel.ForEach()	620
Вызов множества методов с помощью метода Parallel.Invoke()	620
Задачи	621
Запуск задач	621
Получение результатов из задач	623
Задачи продолжения	624
Иерархии задач	625
Инфраструктура отмены	626
Отмена метода Parallel.For()	626
Отмена задач	627
Пулы потоков	629
Класс Thread	630
Передача данных в потоки	631
Фоновые потоки	632
Приоритет потока	633
Управление потоками	634
Проблемы, связанные с потоками	634
Состязания	634
Взаимоблокировки	637
Синхронизация	639
Оператор lock и безопасность в отношении потоков	639
Класс Interlocked	644
Класс Monitor	645

Структура SpinLock	646
Класс WaitHandle	646
Класс Mutex	647
Семафор	648
События	650
Класс Barrier	653
Класс ReaderWriterLockSlim	654
Таймеры	657
Поток данных	658
Использование блока действия	659
Блоки источника и приемника	659
Соединение блоков	660
Резюме	662
Глава 22. Безопасность	664
Введение	665
Аутентификация и авторизация	665
Идентификационные данные и принципалы	665
Роли	667
Декларативная безопасность на основе ролей	667
Заявки	668
Службы клиентских приложений	669
Шифрование	674
Подпись	676
Обмен ключами и безопасная передача данных	678
Управление доступом к ресурсам	681
Безопасность доступа кода	684
Модель Security Transparency Level 2	684
Разрешения	685
Распространение кода с использованием сертификатов	690
Резюме	691
Глава 23. Взаимодействие	692
.NET и COM	693
Метаданные	694
Освобождение памяти	694
Интерфейсы	694
Привязка к методам	696
Типы данных	696
Регистрация	696
Многопоточность	697
Обработка ошибок	698
События	698
Маршализация	699
Использование компонента COM в клиенте .NET	699
Создание компонента COM	700
Создание вызываемой оболочки времени выполнения	705
Использование оболочки RCW	706
Использование сервера COM с динамическими расширениями языка	707

Проблемы, связанные с потоками	708
Добавление точек подключения	708
Использование компонента .NET в клиенте COM	710
Вызываемая оболочка COM	711
Создание компонента .NET	711
Создание библиотеки типов	712
Атрибуты, отвечающие за взаимодействие с COM	714
Регистрация объектов COM	716
Создание приложения клиента COM	717
Добавление точек подключения	718
Создание клиента с объектом приемника	719
Вызов платформы	721
Резюме	725
Глава 24. Манипулирование файлами и реестром	726
Файл и реестр	727
Управление файловой системой	727
Классы .NET, представляющие файлы и папки	728
Класс Path	731
Пример FileProperties	731
Перемещение, копирование и удаление файлов	736
Пример FilePropertiesAndMovement	736
Код приложения FilePropertiesAndMovement	737
Чтение и запись данных в файлы	739
Чтение файла	739
Запись в файл	740
Потоки	741
Буферизованные потоки	743
Чтение и запись в двоичные файлы с использованием класса FileStream	744
Чтение и запись в текстовые файлы	748
Файлы, отображаемые в память	754
Чтение информации об устройствах	755
Безопасность файлов	757
Чтение списков ACL для файла	757
Чтение списков ACL для каталога	758
Добавление и удаление списков ACL для файла	759
Чтение и запись в реестр	760
Реестр	761
Классы .NET для работы с реестром	763
Чтение и запись в изолированное хранилище	766
Резюме	769
Глава 25. Транзакции	770
Введение	771
Обзор	771
Фазы транзакции	772
Свойства ACID	772
Базы данных и классы сущностей	773
Традиционные транзакции	775

Транзакции ADO.NET	775
Пространство имен System.EnterpriseServices	776
Пространство имен System.Transactions	777
Фиксируемые транзакции	778
Продвижение транзакций	780
Зависимые транзакции	782
Охватывающие транзакции	784
Уровень изоляции	790
Специальные диспетчеры ресурсов	791
Транзакционные ресурсы	793
Транзакции файловой системы	797
Резюме	801
Глава 26. Работа с сетью	802
Взаимодействие с сетью	803
Класс WebClient	803
Загрузка файлов	804
Базовый пример WebClient	804
Выгрузка файлов	805
Классы WebRequest и WebResponse	805
Аутентификация	807
Работа с прокси-серверами	808
Асинхронные запросы страниц	808
Отображение вывода как HTML-страницы	809
Обеспечение простого просмотра веб-страниц из приложения	809
Запуск экземпляров Internet Explorer	811
Предоставление приложению дополнительных средств Internet Explorer	811
Вывод на печать с использованием элемента управления WebBrowser	816
Отображение кода запрошенной страницы	816
Иерархия классов WebRequest и WebResponse	817
Служебные классы	818
URI	818
IP-адреса и имена DNS	819
Низкоуровневые протоколы	821
Использование Smtplib	823
Использование классов TCP	824
Примеры TcpSend и TcpReceive	824
Сравнение TCP и UDP	826
Класс UdpClient	827
Класс Socket	827
Веб-сокеты	831
Резюме	834
Глава 27. Службы Windows	835
Что собой представляет служба Windows	836
Архитектура служб Windows	836
Программа службы	837
Программа для управления службой	838
Программа для конфигурирования службы	838
Классы для служб Windows	839

Создание программы службы Windows	839
Создание основной функциональности для службы	840
Пример QuoteClient	843
Программа службы Windows	846
Многопоточность и службы	850
Установка службы	850
Программа установки	850
Класс Installer	851
Классы ProcessInstaller и ServiceInstaller	851
Класс ServiceInstallerDialog	854
Мониторинг и управление службами Windows	854
Оснастка Services консоли MMC	855
Утилита net.exe	855
Утилита sc.exe	855
Окно Server Explorer в Visual Studio	856
Создание специального класса ServiceController	856
Мониторинг службы	856
Выявление неисправностей и регистрация событий	864
Резюме	865
Глава 28. Локализация	866
Глобальные рынки	867
Пространство имен System.Globalization	867
Особенности кодировки Unicode	867
Культуры и регионы	868
Культуры в действии	872
Сортировка	878
Ресурсы	879
Создание ресурсных файлов	879
Утилита генерации ресурсных файлов	880
Класс ResourceWriter	880
Использование ресурсных файлов	881
Пространство имен System.Resources	885
Локализация приложений Windows Forms с помощью Visual Studio	885
Изменение культуры программным образом	890
Использование специальных ресурсных сообщений	891
Автоматический поиск запасных вариантов для ресурсов	892
Возможность получения переводов из внешних источников	892
Локализация приложений ASP.NET Web Forms	893
Локализация приложений WPF	895
Использование ресурсов .NET в WPF	895
Словари ресурсов XAML	896
Специальный класс для чтения ресурсов	900
Создание класса DatabaseResourceReader	900
Создание класса DatabaseResourceSet	902
Создание класса DatabaseResourceManager	902
Создание клиентского приложения для DatabaseResourceReader	903
Создание специальных культур	903
Локализация приложений Windows Store	905

Использование ресурсов	905
Локализация с помощью набора средств для многоязычных приложений	906
Резюме	907
Глава 29. Основы XAML	908
Использование XAML	909
Понятие XAML	909
Отображение элементов на объекты .NET	910
Использование специальных классов .NET	911
Свойства как атрибуты	912
Свойства как элементы	913
Важные типы .NET	913
Использование коллекций в XAML	913
Вызов конструкторов в коде XAML	914
Свойства зависимости	914
Создание свойства зависимости	915
Обратный вызов приведения значений	916
Обратные вызовы и события изменения значений	917
Пузырьковое и туннельное распространение событий	918
Присоединяемые свойства	921
Расширения разметки	923
Создание специальных расширений разметки	923
Расширения разметки, определенные в XAML	925
Чтение и запись XAML	925
Резюме	927
Глава 30. Инфраструктура Managed Extensibility Framework	928
Введение	929
Архитектура MEF	929
Использование атрибутов MEF	930
Регистрация частей на основе соглашений	936
Определение контрактов	937
Экспортирование частей	939
Создание частей	939
Экспортирование свойств и методов	943
Экспортирование метаданных	945
Использование метаданных для отложенной загрузки	947
Импортирование частей	948
Импортирование коллекций	949
Отложенная загрузка частей	951
Чтение метаданных из частей, создаваемых отложенным образом	951
Контейнеры и поставщики экспорта	953
Каталоги	956
Резюме	957
Глава 31. Windows Runtime	958
Обзор	959
Сравнение .NET и Windows Runtime	959
Пространства имен	959

Метаданные	962
Языковые проекции	962
Типы Windows Runtime	965
Компоненты Windows Runtime	966
Коллекция	966
Потоки	967
Делегаты и события	968
Асинхронные операции	968
Приложения Windows 8	969
Жизненный цикл приложений	971
Состояния выполнения приложения	972
Класс <code>SuspensionManager</code>	973
Состояние навигации	974
Тестирование приостановки	975
Состояние страницы	975
Параметры приложения	976
Возможности веб-камеры	979
Резюме	980
Часть IV. Данные	983
Глава 32. Ядро ADO.NET	984
Обзор ADO.NET	985
Пространства имен	985
Разделяемые классы	986
Классы, специфичные для базы данных	987
Использование подключений к базе данных	988
Управление строками подключений	989
Эффективное использование подключений	990
Транзакции	992
Команды	994
Выполнение команд	995
Вызов хранимых процедур	998
Быстрый доступ к данным: объект чтения данных	1000
Асинхронный доступ к данным: использование класса <code>Task</code> и ключевого слова <code>await</code>	1003
Управление данными и отношениями: класс <code>DataSet</code>	1005
Таблицы данных	1006
Отношения между данными	1012
Ограничения данных	1013
Схемы XML: генерация кода с помощью XSD	1016
Заполнение набора данных	1022
Заполнение <code>DataSet</code> с помощью адаптера данных	1022
Заполнение <code>DataSet</code> из XML	1023
Сохранение изменений в наборе данных	1023
Обновление с помощью адаптеров данных	1023
Запись вывода XML	1026
Работа с ADO.NET	1027
Многоуровневая разработка	1027

Генерация ключей в SQL Server	1029
Соглашения об именовании	1031
Резюме	1032
Глава 33. ADO.NET Entity Framework	1034
Программирование с применением Entity Framework	1035
Отображение в Entity Framework	1036
Логический уровень	1037
Концептуальный уровень	1038
Уровень отображения	1040
Строка подключения	1040
Сущности	1041
Объектный контекст	1045
Отношения	1047
Таблица на иерархию	1047
Таблица на тип	1049
Ленивая, отложенная и немедленная загрузка	1050
Запрашивание данных	1051
Язык Entity SQL	1051
Объектный запрос	1053
Технология LINQ to Entities	1055
Запись данных в базу данных	1056
Отслеживание объектов	1056
Информация об изменениях	1057
Присоединение и отсоединение сущностей	1059
Сохранение изменений сущностей	1059
Использование объектов POCO	1060
Определение сущностных типов	1060
Создание контекста данных	1061
Запросы и обновления	1062
Использование модели программирования “сначала код”	1062
Определение сущностных типов	1062
Создание контекста данных	1063
Создание базы данных и хранение сущностей	1063
База данных	1064
Запрос данных	1065
Индивидуальная настройка генерации базы данных	1065
Резюме	1067
Глава 34. Работа с XML	1068
Язык XML	1069
Поддержка стандартов XML в .NET	1069
Знакомство с пространством имен System.Xml	1070
Использование классов из пространства имен System.Xml	1071
Чтение и запись потоков данных XML	1071
Использование класса XmlReader	1072
Проверка достоверности в XmlReader	1076
Использование класса XmlWriter	1077
Использование модели DOM в .NET	1079

Использование класса XmlDocument	1080
Использование класса XPathNavigator	1084
Пространство имен System.Xml.XPath	1084
Пространство имен System.Xml.Xsl	1089
Использование XsltArgumentList	1091
XML и ADO.NET	1095
Преобразование данных ADO.NET в XML	1095
Преобразование данных XML в формат ADO.NET	1100
Сериализация объектов в XML	1102
Сериализация при отсутствии доступа к исходному коду	1109
LINQ to XML и .NET	1111
Работа с различными объектами XML	1111
Класс XmlDocument	1112
Класс XElement	1112
Класс XNamespace	1113
Класс XComment	1115
Класс XAttribute	1115
Использование LINQ для запрашивания XML-документов	1117
Запрашивание статических документов XML	1117
Запрашивание динамических документов XML	1118
Другие приемы запрашивания XML-документов	1119
Чтение данных из XML-документа	1119
Запись данных в XML-документ	1120
Резюме	1122
Часть V. Презентация	1123
Глава 35. Ядро WPF	1124
Обзор	1125
Пространства имен	1125
Иерархия классов	1127
Фигуры	1128
Геометрические объекты	1129
Трансформация	1132
Кисти	1133
SolidColorBrush	1134
LinearGradientBrush	1134
RadialGradientBrush	1134
DrawingBrush	1135
ImageBrush	1135
VisualBrush	1136
Элементы управления	1137
Простые элементы управления	1137
Элементы управления содержимым	1138
Элементы управления с озаглавленным содержимым	1139
Элементы ItemsControl	1140
Элементы HeaderedItemsControl	1140
Декорация	1141

Компоновка	1142
StackPanel	1142
WrapPanel	1142
Canvas	1143
DockPanel	1143
Grid	1144
Стили и ресурсы	1145
Стили	1145
Ресурсы	1147
Системные ресурсы	1148
Доступ к ресурсам из кода	1148
Динамические ресурсы	1149
Словари ресурсов	1149
Триггеры	1151
Триггеры свойств	1151
MultiTrigger	1152
Триггеры данных	1153
Шаблоны	1154
Шаблоны элементов управления	1155
Шаблоны данных	1158
Стилизация элемента управления ListBox	1159
Шаблон ItemTemplate	1160
Шаблоны элементов управления для элементов ListBox	1161
Анимация	1163
Timeline	1163
Нелинейная анимация	1166
Триггеры событий	1167
Анимация с помощью ключевых кадров	1169
Диспетчер визуальных состояний	1170
Визуальные состояния	1171
Переходы	1173
Трехмерная графика	1173
Модель	1174
Камеры	1176
Источники света	1176
Вращение	1176
Резюме	1177
Глава 36. Разработка бизнес-приложений с помощью WPF	1178
Введение	1179
Меню и ленточные элементы управления	1179
Элементы управления меню	1179
Ленточные элементы управления	1180
Использование команд	1183
Определение команд	1183
Определение источников команд	1184
Привязки команд	1185
Привязка данных	1185
Содержимое приложения BooksDemo	1186

Привязка с помощью XAML	1187
Привязка к простым объектам	1190
Уведомление об изменении	1192
Поставщик объектов данных	1194
Привязка к спискам	1196
Привязка типа “главная–детали”	1199
Множественная привязка	1199
Привязка с приоритетами	1201
Преобразование значений	1203
Динамическое добавление элементов списка	1204
Динамическое добавление элементов вкладок	1205
Селектор шаблонов данных	1206
Привязка к XML	1208
Проверка достоверности во время привязки и обработка ошибок	1210
Элемент управления TreeView	1217
Элемент управления DataGrid	1221
Специальные столбцы	1223
Row Details	1224
Группирование с помощью DataGrid	1224
Динамическое формирование	1226
Резюме	1232
Глава 37. Создание документов с помощью WPF	1233
Введение	1234
Текстовые элементы	1234
Шрифты	1234
TextEffect	1235
Inline	1237
Block	1238
List	1239
Table	1240
Анкеры для блоков	1241
Документы нефиксированного формата	1242
Документы фиксированного формата	1246
Документы XPS	1250
Печать	1251
Печать с помощью PrintDialog	1252
Печать визуальных элементов	1252
Резюме	1254
Глава 38. Приложения Windows Store	1255
Обзор	1256
Современный дизайн пользовательского интерфейса в Windows 8	1256
Контент, а не оформление	1256
Быстрота и плавность	1257
Читабельность	1258
Основная функциональность примера приложения	1258
Файлы и папки	1259
Данные приложения	1260

Страницы приложения	1264
Панели приложения	1269
Запуск и навигация	1270
Изменения компоновки	1273
Хранилище	1276
Определение контракта данных	1276
Запись перемещаемых данных	1278
Чтение данных	1279
Запись изображений	1280
Чтение изображений	1282
Средства выбора	1283
Контракт разделения	1284
Общий источник	1284
Общий приемник	1286
Плитки	1288
Резюме	1290
Глава 39. Основы ASP.NET	1291
Платформа .NET Framework для веб-приложений	1292
ASP.NET Web Forms	1292
ASP.NET Web Pages	1293
ASP.NET MVC	1293
Веб-технологии	1294
HTML	1294
CSS	1294
JavaScript и jQuery	1295
Хостинг и конфигурация	1296
Обработчики и модули	1298
Создание специального обработчика	1299
Обработчики ASP.NET	1300
Создание специального модуля	1301
Общие модули	1302
Глобальный класс приложения	1303
Запрос и ответ	1304
Использование объекта <code>HttpRequest</code>	1304
Использование объекта <code>HttpResponse</code>	1306
Управление состоянием	1306
Состояние представления	1306
Cookie-наборы	1307
Сеанс	1309
Приложение	1311
Кеш	1312
Профили	1313
Членство и роли	1317
Конфигурирование членства	1318
Использование Membership API	1319
Включение Roles API	1320
Резюме	1321

Глава 40. ASP.NET Web Forms	1322
Обзор	1323
Модель страницы ASPX	1323
Добавление элементов управления	1324
Использование событий	1325
Работа с обратными отправками	1325
Использование автоматических обратных отправок	1326
Выполнение обратных отправок другим страницам	1326
Определение строго типизированных межстраничных обратных отправок	1327
Использование событий страницы	1328
Код внутри страницы ASPX	1329
Серверные элементы управления	1332
Мастер-страницы	1333
Создание мастер-страницы	1333
Использование мастер-страниц	1334
Выборочное определение содержимого мастер-страницы из страницы содержимого	1336
Навигация	1337
Карта сайта	1337
Элемент управления Menu	1338
Путь в меню	1339
Проверка достоверности пользовательского ввода	1339
Использование элементов управления проверкой достоверности	1339
Использование сводки по проверке достоверности	1340
Группы проверки достоверности	1341
Доступ к данным	1341
Использование Entity Framework	1343
Использование источника сущностных данных	1343
Сортировка и редактирование	1346
Настройка столбцов	1346
Использование шаблонов с сеткой	1347
Настройка создания объектного контекста	1349
Класс ObjectDataSource	1350
Безопасность	1350
Включение аутентификации с помощью форм	1351
Элементы управления входом	1351
Ajax	1352
Что такое ASP.NET AJAX?	1354
Пример веб-сайта ASP.NET AJAX	1357
Конфигурирование веб-сайта, оснащенного ASP.NET AJAX	1360
Добавление функциональности ASP.NET AJAX	1361
Резюме	1368
Глава 41. ASP.NET MVC	1369
Обзор ASP.NET MVC	1370
Определение маршрутов	1371
Добавление маршрутов	1372
Ограничения маршрутов	1372

Создание контроллеров	1373
Методы действий	1374
Параметры	1374
Возвращение данных	1375
Создание представлений	1376
Передача данных в представления	1378
Синтаксис Razor	1378
Строго типизированные представления	1379
Компоновка	1380
Частичные представления	1383
Отправка данных со стороны клиента	1386
Связыватель модели	1387
Аннотации и проверка достоверности	1388
Вспомогательные методы HTML	1389
Простые вспомогательные методы	1390
Использование данных модели	1390
Определение атрибутов HTML	1391
Создание списков	1392
Строго типизированные вспомогательные методы	1393
Расширения редактора	1393
Создание специальных вспомогательных методов	1394
Шаблоны	1394
Создание приложения, управляемого данными	1395
Определение модели	1396
Создание контроллеров и представлений	1396
Фильтры действий	1401
Аутентификация и авторизация	1403
Модель для входа	1403
Контроллер для входа	1403
Представление для входа	1404
ASP.NET Web API	1405
Доступ к данным с использованием модели программирования	
“сначала код” в Entity Framework	1406
Определение маршрутов для ASP.NET Web API	1407
Реализация контроллера	1407
Клиентское приложение, использующее jQuery	1409
Резюме	1411
Глава 42. ASP.NET Dynamic Data	1412
Обзор	1413
Создание веб-приложений динамических данных	1413
Конфигурирование механизма формирования шаблонов	1414
Просмотр результата	1415
Настройка веб-сайтов с динамическими данными	1417
Управление механизмом формирования шаблонов	1418
Настройка шаблонов	1419
Конфигурирование маршрутизации	1424
Резюме	1425

Часть VI. Коммуникации (на веб-сайте)

Глава 43. Windows Communication Foundation веб-сайт

Глава 44. Службы данных WCF веб-сайт

Глава 45. Windows Workflow Foundation веб-сайт

Глава 46. Одноранговые сети веб-сайт

Глава 47. Организация очередей сообщений веб-сайт

Предметный указатель 1426

13

Асинхронное программирование

В ЭТОЙ ГЛАВЕ...

- Важность асинхронного программирования
- Асинхронные шаблоны
- Ключевые слова `async` и `await`
- Создание и использование асинхронных методов
- Обработка ошибок с помощью асинхронных методов

Загружаемый код для этой главы

Загружаемый код для этой главы содержит следующие основные примеры:

- `AsyncPatterns`
- `Foundations`
- `ErrorHandling`

Важность асинхронного программирования

Наиболее важным изменением версии C# 5 является прогресс в отношении асинхронного программирования. В C# 5 добавлены всего два ключевых слова: `async` и `await`. На этих ключевых словах и сосредоточено основное внимание в этой главе.

При *асинхронном программировании* вызванный метод выполняется в фоновом режиме (обычно с помощью потока или задачи), а вызывающий поток не блокируется.

В этой главе вы ознакомитесь с различными шаблонами асинхронного программирования, такими как *асинхронный шаблон*, *асинхронный шаблон, основанный на событиях*, и новый *асинхронный шаблон, основанный на задачах* (task-based asynchronous pattern – TAP). Шаблон TAP использует ключевые слова `async` и `await`. Сравнивая указанные шаблоны, можно оценить реальные преимущества нового стиля асинхронного программирования.

После обсуждения шаблонов будет продемонстрирована основа асинхронного программирования на примере создания задач и вызова асинхронных методов. Вы узнаете, что находится “за кулисами” задач продолжения и контекста синхронизации.

Обработка ошибок заслуживает особого внимания; как и в случае асинхронных задач, некоторые сценарии требуют разных подходов к обработке ошибок.

В конце этой главы обсуждаются способы обеспечения отмены. Фоновые задачи могут требовать некоторого времени на свое выполнение, поэтому иногда возникает необходимость в отмене задачи, пока она еще выполняется. В настоящей главе будет показано, как это делать.

Другие сведения о параллельном программировании приведены в главе 21.

Пользователей обычно раздражает, если приложение не реагирует немедленно на запросы. Работая с мышью, мы уже привыкли к возникновению задержек, т.к. сталкивались с подобным поведением в течение нескольких десятилетий. Однако при наличии сенсорного пользовательского интерфейса приложение должно реагировать на запросы немедленно. В противном случае пользователь попытается повторить действие.

Поскольку в старых версиях .NET Framework асинхронное программирование было затруднено, оно не всегда делалось, когда это было необходимо. Примером приложения, которое довольно часто блокировало поток пользовательского интерфейса, может служить Visual Studio 2010. В этой версии открытие решения, содержащего сотни проектов, приводило к длительной паузе, в течение которой вполне можно было успеть выпить чашку кофе. В Visual Studio 2012 подобное не происходит, т.к. проекты загружаются в фоновом режиме, с загрузкой первым выбранного проекта. Такое поведение загрузки – лишь один пример важных изменений, связанных с асинхронным программированием, которые были внесены в версию Visual Studio 2012. Аналогичным образом пользователи Visual Studio 2010 очень хорошо знакомы с ситуацией, когда какое-то диалоговое окно не реагирует на действия. В Visual Studio 2012 это менее вероятно.

Многие API-интерфейсы в .NET Framework предлагают синхронные и асинхронные версии. Так как синхронную версию API-интерфейса намного легче использовать, ее часто применяли там, где это совершенно не подходило. Благодаря новой исполняющей среде Windows Runtime (WinRT), если ожидается, что вызов API-интерфейса займет более 40 миллисекунд, будет доступна только асинхронная версия этого API-интерфейса. С появлением .NET 4.5 асинхронное программирование стало таким же простым, как синхронное, поэтому никаких препятствий к использованию асинхронных API-интерфейсов возникать не должно.

Асинхронные шаблоны

Перед тем, как перейти к рассмотрению новых ключевых слов `async` и `await`, имеет смысл разобраться в асинхронных шаблонах .NET Framework. Асинхронные возможности были доступны, начиная еще с версии .NET 1.0, и многие классы в .NET Framework реализуют один или несколько таких шаблонов. Асинхронный шаблон также доступен с типом делегата.

Из-за того, что обновление пользовательского интерфейса с помощью как Windows Forms, так и WPF с асинхронным шаблоном было довольно сложным, в версии .NET 2.0 появился *асинхронный шаблон, основанный на событиях*. В данном шаблоне обработчик событий вызывается из потока, который владеет контекстом синхронизации, поэтому код обновления пользовательского интерфейса здесь прост. Ранее этот шаблон был известен также под названием *шаблона асинхронных компонентов*.

В .NET 4.5 введен новый подход к асинхронному программированию — *асинхронный шаблон, основанный на задачах* (TAP). Этот шаблон базируется на типе `Task`, появившемся в .NET 4, и применении ключевых слов `async` и `await`.

Чтобы оценить преимущество использования ключевых слов `async` и `await`, в первом примере приложения для предоставления обзора асинхронного программирования применяется инфраструктура Windows Presentation Foundation (WPF) и работа с сетью. Если вы не имеете опыта использования WPF и программирования для сети, переживать не следует. Вы все равно сможете уловить суть и понять принципы асинхронного программирования.

В последующих примерах демонстрируются отличия между асинхронными шаблонами. Затем с помощью простых консольных приложений будут проиллюстрированы основы асинхронного программирования.

На заметку! Инфраструктура WPF подробно рассматривается в главах 35 и 36, а программирование для сети — в главе 26.

Для отражения отличий между асинхронными шаблонами создается WPF-приложение, в котором используются типы из библиотеки классов. Это приложение позволяет находить изображения в веб-сети с применением служб Bing и Flickr. Для нахождения изображений пользователь вводит критерий поиска, который отправляется службам Bing и Flickr в виде простого HTTP-запроса.

Внешний вид пользовательского интерфейса, построенного в Visual Studio, можно видеть на рис. 13.1. В верхней части расположено поле ввода, за которым следуют несколько кнопок, предназначенных для запуска поиска и очистки списка результатов. Слева под управляющей областью находится элемент управления `ListBox`, в котором будут отображаться все найденные изображения. Справа помещен элемент управления `Image`, который позволяет показывать изображение, выбранное в `ListBox`, с высоким разрешением.

Мы начнем с библиотеки классов `AsyncLib`, содержащей множество вспомогательных классов. Эти классы используются WPF-приложением.

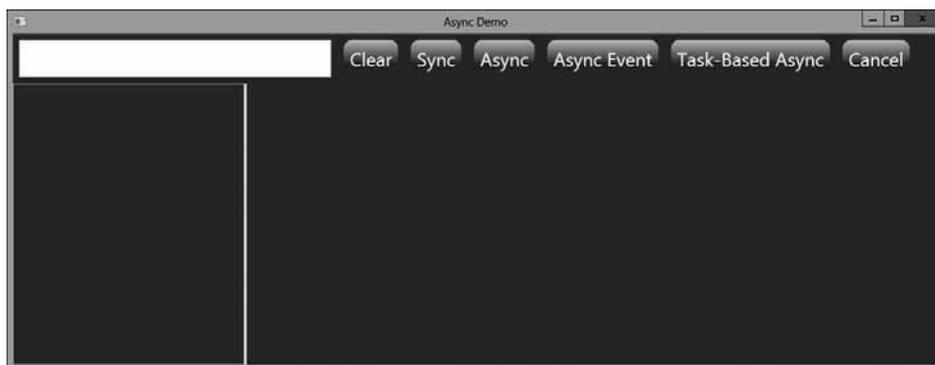


Рис. 13.1. Пользовательский интерфейс примера приложения

Класс `SearchItemResult` представляет одиночный элемент из результирующей коллекции, который используется для вывода изображения вместе с заголовком и источником, откуда было получено изображение. В этом классе просто определены простые свойства: `Title`, `Url`, `ThumbnailUrl` и `Source`. Свойство `ThumbnailUrl` служит для ссылки на изображение-миниатюру, свойство `Url` содержит ссылку на изображение крупного размера. Свойство `Title` хранит текст, описывающий изображение. Базовым классом для `SearchItemResult` является `BindableBase`. Этот базовый класс всего лишь поддерживает механизм уведомлений за счет реализации интерфейса `INotifyPropertyChanged`, который применяется инфраструктурой WPF для организации обновлений через привязку данных (файл `AsyncLib/SearchItemResult.cs`):

```
namespace Wrox.ProCSharp.Async
{
    public class SearchItemResult : BindableBase
    {
        private string title;
        public string Title
        {
            get { return title; }
            set { SetProperty(ref title, value); }
        }
        private string url;
        public string Url
        {
            get { return url; }
            set { SetProperty(ref url, value); }
        }
        private string thumbnailUrl;
        public string ThumbnailUrl
        {
            get { return thumbnailUrl; }
            set { SetProperty(ref thumbnailUrl, value); }
        }
        private string source;
        public string Source
        {
            get { return source; }
            set { SetProperty(ref source, value); }
        }
    }
}
```

Еще одним классом, используемым с привязкой данных, является `SearchInfo`. Его свойство `SearchTerm` содержит пользовательский ввод для поиска изображений указанного типа. Свойство `List` возвращает список всех найденных изображений, представленных с помощью типа `SearchItemResult` (файл `AsyncLib/SearchInfo.cs`):

```
using System.Collections.ObjectModel;
namespace Wrox.ProCSharp.Async
{
    public class SearchInfo : BindableBase
    {
        public SearchInfo()
        {
            list = new ObservableCollection<SearchItemResult>();
            list.CollectionChanged += delegate { OnPropertyChanged("List"); };
        }
        private string searchTerm;
        public string SearchTerm
        {
            get { return searchTerm; }
            set { SetProperty(ref searchTerm, value); }
        }
    }
}
```

```

private ObservableCollection<SearchItemResult> list;
public ObservableCollection<SearchItemResult> List
{
    get
    {
        return list;
    }
}
}
}

```

В коде XAML для ввода критерия поиска применяется `TextBox`. Этот элемент управления привязан к свойству `SearchTerm` типа `SearchInfo`. Элементы управления `Button` запускают обработчики событий, например, кнопка `Sync` (Синхронизировать) вызывает метод `OnSearchSync()` (файл `AsyncPatterns/MainWindow.xaml`):

```

<StackPanel Orientation="Horizontal" Grid.Row="0">
    <StackPanel.LayoutTransform>
        <ScaleTransform ScaleX="2" ScaleY="2" />
    </StackPanel.LayoutTransform>
    <TextBox Text="{Binding SearchTerm}" Width="200" Margin="4" />
    <Button Click="OnClear">Clear</Button>
    <Button Click="OnSearchSync">Sync</Button>
    <Button Click="OnSearchAsyncPattern">Async</Button>
    <Button Click="OnAsyncEventPattern">Async Event</Button>
    <Button Click="OnTaskBasedAsyncPattern">Task Based Async</Button>
</StackPanel>

```

Вторая часть кода XAML содержит элемент управления `ListBox`. Для создания специального представления элементов в `ListBox` применяется шаблон `ItemTemplate`. Каждый элемент списка представлен с помощью двух элементов управления `TextBlock` и одного `Image`. Элемент управления `ListBox` привязан к свойству `List` класса `SearchInfo`, а свойства элементов управления для элемента списка привязаны к свойствам типа `SearchItemResult`:

```

<Grid Grid.Row="1">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="3*" />
    </Grid.ColumnDefinitions>
    <ListBox Grid.IsSharedSizeScope="True" ItemsSource="{Binding List}"
        Grid.Column="0" IsSynchronizedWithCurrentItem="True"
        Background="Black">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <Grid>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition SharedSizeGroup="ItemTemplateGroup" />
                    </Grid.ColumnDefinitions>
                    <StackPanel HorizontalAlignment="Stretch" Orientation="Vertical"
                        Background="{StaticResource linearBackgroundBrush}">
                        <TextBlock Text="{Binding Source}" Foreground="White" />
                        <TextBlock Text="{Binding Title}" Foreground="White" />
                        <Image HorizontalAlignment="Center"
                            Source="{Binding ThumbnailUrl}" Width="100" />
                    </StackPanel>
                </Grid>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
    <GridSplitter Grid.Column="1" Width="3" HorizontalAlignment="Left" />
    <Image Grid.Column="1" Source="{Binding List/Url}" />
</Grid>

```

А теперь давайте займемся анализом класса `BingRequest`. Этот класс содержит некоторую информацию о том, как отправлять запрос службе Bing. Свойство `Url` данного класса возвращает строку URL, которая может применяться для запрашивания изображений. Запрос состоит из критерия поиска, количества изображений, которые должны быть запрошены (`Count`), и количества изображений, которые должны быть пропущены (`Offset`). Для службы Bing требуется аутентификация. Идентификатор пользователя определен в `AppId`, и он применяется со свойством `Credentials`, которое возвращает объект `NetworkCredential`. Для успешного запуска приложения необходимо зарегистрироваться в магазине Windows Azure Marketplace и подписаться на Bing Search API. На момент написания этой книги первые 5000 транзакций в месяц были бесплатными — этого должно хватить для работы с примером приложения. Каждый поиск представляет собой одну транзакцию. Подписка на Bing Search API производится по адресу <https://datamarket.azure.com/dataset/bing/search>. После регистрации понадобится скопировать идентификатор приложения. Этот идентификатор приложения необходимо добавить в класс `BingRequest`.

В результате отправки запроса к Bing с использованием созданного URL служба Bing возвращает разметку XML. Метод `Parse()` класса `BingRequest` позволяет выполнить разбор этой разметки XML и возвращает коллекцию объектов `SearchItemResult` (файл `AsyncLib/BingRequest.cs`):

На заметку! Методы `Parse()` в классах `BingRequest` и `FlickrRequest` пользуются LINQ to XML. Технология LINQ to XML описана в главе 34.

```
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Xml.Linq;

namespace Wrox.ProCSharp.Async
{
    public class BingRequest : IImageRequest
    {
        private const string AppId = "enter your Bing AppId here";

        public BingRequest()
        {
            Count = 50;
            Offset = 0;
        }

        private string searchTerm;
        public string SearchTerm
        {
            get { return searchTerm; }
            set { searchTerm = value; }
        }

        public ICredentials Credentials
        {
            get
            {
                return new NetworkCredentials(AppId, AppId);
            }
        }

        public string Url
        {
            get
            {
```

```

        return string.Format("https://api.datamarket.azure.com/" +
            "Data.ashx/Bing/Search/v1/Image?Query=%27{0}%27&" +
            "$top={1}&$skip={2}&$format=Atom",
            SearchTerm, Count, Offset);
    }
}

public int Count { get; set; }
public int Offset { get; set; }

public IEnumerable<SearchItemResult> Parse(string xml)
{
    XElement respXml = XElement.Parse(xml);
    // XNamespace atom = XNamespace.Get("http://www.w3.org/2005/Atom");
    XNamespace d = XNamespace.Get(
        "http://schemas.microsoft.com/ado/2007/08/dataservices");
    XNamespace m = XNamespace.Get(
        "http://schemas.microsoft.com/ado/2007/08/dataservices/metadata");
    return (from item in respXml.Descendants(m + "properties")
        select new SearchItemResult
        {
            Title = new string(item.Element(d +
                "Title").Value.Take(50).ToArray()),
            Url = item.Element(d + "MediaUrl").Value,
            ThumbnailUrl = item.Element(d + "Thumbnail").
                Element(d + "MediaUrl").Value,
            Source = "Bing"
        }).ToList();
    }
}
}

```

Классы `BingRequest` и `FlickrRequest` реализуют интерфейс `IImageRequest`. В этом интерфейсе определены свойства `SearchTerm` и `Url`, а также метод `Parse()`, который позволяет организовать простую итерацию по обоим поставщикам служб изображений (файл `AsyncLib/IImageRequest.cs`):

```

using System;
using System.Collections.Generic;
using System.Net;

namespace Wrox.ProCSharp.Async
{
    public interface IImageRequest
    {
        string SearchTerm { get; set; }
        string Url { get; }
        IEnumerable<SearchItemResult> Parse(string xml);
        ICredentials Credentials { get; }
    }
}

```

Класс `FlickrRequest` очень похож на `BingRequest`. Он только создает другой URL для запрашивания изображений по критерию поиска и содержит собственную реализацию метода `Parse()`, поскольку разметка XML, возвращаемая службой Flickr, отличается от разметки, которую возвращает Bing. Как и в случае Bing, чтобы создать идентификатор приложения для Flickr, необходимо зарегистрироваться в службе Flickr и запросить его: <http://www.flickr.com/services/apps/create/apply/>.

```

using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

```

```

namespace Wrox.ProCSharp.Async
{
    public class FlickrRequest : IImageRequest
    {
        private const string AppId = "Enter your Flickr AppId here";
        public FlickrRequest()
        {
            Count = 50;
            Page = 1;
        }

        private string searchTerm;
        public string SearchTerm
        {
            get { return searchTerm; }
            set { searchTerm = value; }
        }

        public string Url
        {
            get
            {
                return string.Format("http://api.flickr.com/services/rest?" +
                    "api_key={0}&method=flickr.photos.search&content_type=1&" +
                    "text={1}&per_page={2}&page={3}", AppId, SearchTerm, Count, Page);
            }
        }

        public ICredentials Credentials
        {
            get { return null; }
        }

        public int Count { get; set; }
        public int Page { get; set; }
        public IEnumerable<SearchItemResult> Parse(string xml)
        {
            XElement respXml = XElement.Parse(xml);
            return (from item in respXml.Descendants("photo")
                select new SearchItemResult
                {
                    Title = new string(item.Attribute("title").Value.
                        Take(50).ToArray()),
                    Url = string.Format("http://farm{0}.staticflickr.com/" +
                        "{1}/{2}_{3}_z.jpg",
                        item.Attribute("farm").Value, item.Attribute("server").Value,
                        item.Attribute("id").Value, item.Attribute("secret").Value),
                    ThumbnailUrl = string.Format("http://farm{0}." +
                        "staticflickr.com/{1}/{2}_{3}_t.jpg",
                        item.Attribute("farm").Value,
                        item.Attribute("server").Value,
                        item.Attribute("id").Value,
                        item.Attribute("secret").Value),
                    Source = "Flickr"
                }).ToList();
        }
    }
}

```

Теперь нужно просто соединить типы из библиотеки и WPF-приложение. В конструкторе класса `MainWindow` создается экземпляр `SearchInfo` и затем присваивается свойству `DataContext` окна. Привязка данных, показанная ранее в коде XAML, начинает действовать (файл `AsyncPatterns/MainWindow.xaml.cs`):

```
public partial class MainWindow : Window
{
    private SearchInfo searchInfo;

    public MainWindow()
    {
        InitializeComponent();
        searchInfo = new SearchInfo();
        this.DataContext = searchInfo;
    }
}
```

Класс `MainWindow` также содержит вспомогательный метод `GetSearchRequests()`, который возвращает коллекцию объектов `IImageRequest` в форме типов `BingRequest` и `FlickrRequest`. В случае если вы зарегистрировались только в одной из этих служб, измените соответствующим образом код. Разумеется, можно также создать типы `IImageRequest` для других служб, например, `Google` или `Yahoo`. Добавьте типы запросов в возвращаемую коллекцию:

```
private IEnumerable<IImageRequest> GetSearchRequests()
{
    return new List<IImageRequest>
    {
        new BingRequest { SearchTerm = searchInfo.SearchTerm },
        new FlickrRequest { SearchTerm = searchInfo.SearchTerm }
    };
}
```

Синхронный вызов

Теперь, когда все должным образом настроено, давайте начнем с синхронного вызова указанных служб. В обработчике событий щелчка на кнопке `Sync`, `OnSearchSync()`, производится проход по всем поисковым запросам, возвращенным из метода `GetSearchRequests()`, и с помощью класса `WebClient` делается HTTP-запрос с применением свойства `Url`. Метод `DownloadString()` блокируется до тех пор, пока не будет получен результат. Результирующая разметка XML присваивается переменной `resp`. Эта разметка XML анализируется с помощью метода `Parse()`, который возвращает коллекцию объектов `SearchItemResult`. Элементы этой коллекции затем добавляются в список, содержащийся внутри `searchInfo` (файл `AsyncPatterns/MainWindow.xaml.cs`):

```
private void OnSearchSync(object sender, RoutedEventArgs e)
{
    foreach (var req in GetSearchRequests())
    {
        var client = new WebClient();
        client.Credentials = req.Credentials;
        string resp = client.DownloadString(req.Url);
        IEnumerable<SearchItemResult> images = req.Parse(resp);
        foreach (var image in images)
        {
            searchInfo.List.Add(image);
        }
    }
}
```

В функционирующем приложении (рис. 13.2) пользовательский интерфейс блокируется до тех пор, пока метод `OnSearchSync()` не завершит обращения через сеть к службам `Bing` и `Flickr`, а также анализ результатов. Промежуток времени, необходимый для завершения этих вызовов, варьируется в зависимости от пропускной способности сети и текущей рабочей нагрузки служб `Bing` и `Flickr`. Тем не менее, в любом случае ожидание неприятно для пользователя.



Рис. 13.2. Пример приложения во время выполнения

Итак, давайте реализуем вызов асинхронным образом.

Асинхронный шаблон

Один из способов сделать вызов асинхронным предусматривает использование асинхронного шаблона. При асинхронном шаблоне определяются методы `BeginXXX()` и `EndXXX()`. Например, если предлагается синхронный метод `DownloadString()`, то асинхронными вариантами будут `BeginDownloadString()` и `EndDownloadString()`. Метод `BeginXXX()` принимает все входные аргументы синхронного метода, а `EndXXX()` — все выходные аргументы и возвращаемый тип, чтобы вернуть результат. В асинхронном шаблоне метод `BeginXXX()` также определяет параметр `AsyncCallback`, который принимает делегат, вызываемый по завершении асинхронного метода. Метод `BeginXXX()` возвращает объект `IAsyncResult`, который может применяться для опроса с целью проверки, завершился ли вызов, и ожидания, пока метод закончится.

Класс `WebClient` не предоставляет реализацию асинхронного шаблона. Вместо него мог бы использоваться класс `HttpRequest`, который поддерживает этот шаблон посредством методов `BeginGetResponse()` и `EndGetResponse()`. В следующем примере это не делается, а применяется делегат. Тип делегата определяет метод `Invoke()` для выполнения синхронного вызова метода и методы `BeginInvoke()` и `EndInvoke()` для его использования в асинхронном шаблоне. В примере объявляется делегат `downloadString` типа `Func<string, string>` для ссылки на метод, который принимает параметр `string` и возвращает тип `string`. Метод, на который ссылается переменная `downloadString`, реализован в виде лямбда-выражения и вызывает синхронный метод `DownloadString()` типа `WebClient`. Делегат вызывается асинхронно путем вызова метода `BeginInvoke()`. Для выполнения асинхронного вызова этот метод применяет поток из пула потоков.

Первый параметр метода `BeginInvoke()` — это первый обобщенный параметр `string` делегата `Func`, в котором можно передавать URL. Второй параметр имеет тип `AsyncCallback`. Тип `AsyncCallback` представляет собой делегат, который требует `IAsyncResult` в качестве параметра. Метод, на который ссылаются с помощью этого делегата, вызывается по завершении асинхронного метода. Когда это происходит, вызывается метод `downloadString.EndInvoke()` для извлечения результата, который обрабатывается тем же самым способом, что и описанный ранее — анализ разметки XML и получение коллекции элементов. Однако здесь обратиться напрямую к пользовательскому интерфейсу не удастся, т.к. пользовательский интерфейс привязан к единственному потоку, а метод обратного вызова выполняется внутри фонового потока. Следовательно, необходимо переключиться на поток пользовательского интерфейса с применением свойства `Dispatcher` окна. Метод `Invoke()` класса `Dispatcher` требует в качестве параметра делегат; именно

поэтому указан делегат `Action<SearchItemResult>`, который добавляет элемент в коллекцию, привязанную к пользовательскому интерфейсу (файл `AsyncPatterns/MainWindow.xaml.cs`):

```
private void OnSearchAsyncPattern(object sender, RoutedEventArgs e)
{
    Func<string, ICredentials, string> downloadString = (address, cred) =>
    {
        var client = new WebClient();
        client.Credentials = cred;
        return client.DownloadString(address);
    };
    Action<SearchItemResult> addItem = item => searchInfo.List.Add(item);
    foreach (var req in GetSearchRequests())
    {
        downloadString.BeginInvoke(req.Url, req.Credentials, ar =>
        {
            string resp = downloadString.EndInvoke(ar);
            IEnumerable<SearchItemResult> images = req.Parse(resp);
            foreach (var image in images)
            {
                this.Dispatcher.Invoke(addItem, image);
            }
        }, null);
    }
}
```

Преимущество асинхронного шаблона заключается в том, что он может быть легко реализован с использованием функциональности делегатов. Теперь приложение ведет себя так, как должно; пользовательский интерфейс больше не блокируется. Однако применение асинхронного шаблона сопряжено и с трудностями. К счастью, в версии .NET 2.0 появился асинхронный шаблон, основанный на событиях, который упрощает реализацию обновлений пользовательского интерфейса. Этот шаблон обсуждается в следующем разделе.

На заметку! Типы делегатов и лямбда-выражения рассматривались в главе 8. Потоки и пулы потоков будут описаны в главе 21.

Асинхронный шаблон, основанный на событиях

Асинхронный шаблон, основанный на событиях, используется в методе `OnAsyncEventPattern()`. Данный шаблон реализован классом `WebClient`, следовательно, им можно пользоваться напрямую. Согласно этому шаблону, определяется метод, имя которого заканчивается на `Async`. Таким образом, например, для синхронного метода `DownloadString()` класс `WebClient` предлагает асинхронный вариант `DownloadStringAsync()`. Вместо объявления делегата, вызываемого по завершении асинхронного метода, определяется событие. Событие `DownloadStringCompleted` инициализируется при завершении асинхронного метода `DownloadStringAsync()`. Метод, назначенный в качестве обработчика событий, реализован в виде лямбда-выражения. Реализация очень похожа на предыдущую, но теперь возможно напрямую обращаться к элементам пользовательского интерфейса, т.к. обработчик событий вызывается из потока, имеющего контекст синхронизации, а в случае приложений `Windows Forms` и `WPF` это будет сам поток пользовательского интерфейса (файл `AsyncPatterns/MainWindow.xaml.cs`):

```
private void OnAsyncEventPattern(object sender, RoutedEventArgs e)
{
    foreach (var req in GetSearchRequests())
    {
        var client = new WebClient();
```

```

client.Credentials = req.Credentials;
client.DownloadStringCompleted += (sender1, e1) =>
{
    string resp = e1.Result;
    IEnumerable<SearchItemResult> images = req.Parse(resp);
    foreach (var image in images)
    {
        searchInfo.List.Add(image);
    }
};
client.DownloadStringAsync(new Uri(req.Url));
}
}

```

Преимущество асинхронного шаблона, основанного на событиях, связано с простотой его использования. Однако следует отметить, что этот шаблон не так легко реализовать в специальном классе. Класс `BackgroundWorker` позволяет воспользоваться существующей реализацией этого шаблона для превращения синхронных методов в асинхронные. Упомянутый класс реализует асинхронный шаблон, основанный на событиях.

Это намного упрощает код. Тем не менее, по сравнению с вызовами синхронных методов порядок является обратным. То есть определять, что произойдет, когда вызов метода завершится, нужно перед вызовом асинхронного метода. В следующем разделе мы погрузимся в новый мир асинхронного программирования с применением ключевых слов `async` и `await`.

Асинхронный шаблон, основанный на задачах

В версии .NET 4.5 класс `WebClient` был обновлен, чтобы поддерживать также и асинхронный шаблон, основанный на задачах (TAP). Этот шаблон предполагает определение метода, имя которого заканчивается на `Async`, возвращающего тип `Task`. Поскольку класс `WebClient` уже предлагает метод с суффиксом `Async` для реализации асинхронного шаблона, основанного на событиях, новому методу назначено имя `DownloadStringTaskAsync()`.

Метод `DownloadStringTaskAsync()` объявлен как возвращающий `Task<string>`. Нет никакой необходимости в объявлении переменной типа `Task<string>` для присваивания результата, возвращаемого методом `DownloadStringTaskAsync()`; вместо этого можно объявить переменную типа `string` и применить ключевое слово `await`. Ключевое слово `await` разблокирует поток (в случае потока пользовательского интерфейса) для работы других задач. После того, как метод `DownloadStringTaskAsync()` завершит свою фоновую обработку, поток пользовательского интерфейса сможет продолжить, получив результат из фоновой задачи в строковую переменную `resp`. Выполнение будет продолжено с кода, следующего за этой строкой (файл `AsyncPatterns/MainWindow.xaml.cs`):

```

private async void OnTaskBasedAsyncPattern(object sender,
                                           RoutedEventArgs e)
{
    foreach (var req in GetSearchRequests())
    {
        var client = new WebClient();
        client.Credentials = req.Credentials;
        string resp = await client.DownloadStringTaskAsync(req.Url);
        IEnumerable<SearchItemResult> images = req.Parse(resp);
        foreach (var image in images)
        {
            searchInfo.List.Add(image);
        }
    }
}

```

На заметку! Ключевое слово `async` создает конечный автомат подобно оператору `yield return`, который обсуждался в главе 6.

Код теперь стал намного проще. Отсутствует блокирование, и нет ручного переключения на поток пользовательского интерфейса, т.к. это делается автоматически; к тому же код записывается в том же порядке, что и при синхронном программировании.

Далее код изменяется для использования класса, отличного от `WebClient` — одно из тех, где асинхронный шаблон, основанный на задачах, реализован непосредственно, а синхронные методы вообще не предлагаются. Такой класс появился в .NET 4.5 и называется он `HttpClient`. Асинхронный запрос GET выполняется с помощью метода `GetAsync()`. Для чтения содержимого необходим еще один асинхронный метод. Метод `ReadAsStringAsync()` возвращает содержимое в виде строки.

```
private async void OnTaskBasedAsyncPattern(object sender, RoutedEventArgs e)
{
    foreach (var req in GetSearchRequests())
    {
        var clientHandler = new HttpClientHandler
        {
            Credentials = req.Credentials
        };
        var client = new HttpClient(clientHandler);
        var response = await client.GetAsync(req.Url);
        string resp = await response.Content.ReadAsStringAsync();
        IEnumerable<SearchItemResult> images = req.Parse(resp);
        foreach (var image in images)
        {
            searchInfo.List.Add(image);
        }
    }
}
```

Разбор строки XML может занять некоторое время. Поскольку код разбора выполняется в потоке пользовательского интерфейса, в это время данный поток не может реагировать на запросы, поступающие от пользователя. Чтобы создать из синхронной функциональности фоновую задачу, можно применить метод `Task.Run()`. В показанном ниже примере внутри `Task.Run()` производится разбор строки XML для возврата коллекции `SearchItemResult`:

```
private async void OnTaskBasedAsyncPattern(object sender, RoutedEventArgs e)
{
    foreach (var req in GetSearchRequests())
    {
        var clientHandler = new HttpClientHandler
        {
            Credentials = req.Credentials
        };
        var client = new HttpClient(clientHandler);
        var response = await client.GetAsync(req.Url, cts.Token);
        string resp = await response.Content.ReadAsStringAsync();
        await Task.Run(() =>
        {
            IEnumerable<SearchItemResult> images = req.Parse(resp);
            foreach (var image in images)
            {
                searchInfo.List.Add(image);
            }
        })
    }
}
```

Из-за того, что метод, передаваемый `Task.Run()`, выполняется в фоновом потоке, возникает та же самая проблема с обращением к коду пользовательского интерфейса, которая была описана ранее. Решение могло бы заключаться в вызове `req.Parse()` внутри метода `Task.Run()` и организации цикла `foreach` за пределами задачи для добавления результата к списку в потоке пользовательского интерфейса.

Однако инфраструктура WPF в .NET 4.5 предлагает лучшее решение, которое позволяет заполнять в фоновом потоке коллекции, привязанные к элементам управления пользовательского интерфейса. Это расширение требует только включения синхронизации для коллекции с помощью метода `BindingOperations.EnableCollectionSynchronization()`, как показано в следующем фрагменте кода:

```
public partial class MainWindow : Window
{
    private SearchInfo searchInfo;
    private object lockList = new object();
    public MainWindow()
    {
        InitializeComponent();
        searchInfo = new SearchInfo();
        this.DataContext = searchInfo;
        BindingOperations.EnableCollectionSynchronization(
            searchInfo.List, lockList);
    }
}
```

После демонстрации преимуществ применения ключевых слов `async` и `await` в следующем разделе рассматривается внутренняя поддержка, лежащая в основе этих слов.

Основы асинхронного программирования

Ключевые слова `async` и `await` являются просто средством компилятора. Компилятор создает код, использующий класс `Task`. Вместо применения этих новых ключевых слов ту же функциональность можно получить с помощью C# 4 и методов класса `Task`, но не настолько удобно.

В этом разделе приведена информация о том, что делает компилятор с ключевыми словами `async` и `await`, предложен простой способ создания асинхронного метода, а также показано, как вызвать несколько асинхронных методов параллельно, и каким образом изменить класс, поддерживающий только асинхронный шаблон, для использования новых ключевых слов.

Создание задач

Давайте начнем с синхронного метода `Greeting()`, который делает паузу и затем возвращает строку (файл `Foundations/Program.cs`):

```
static string Greeting(string name)
{
    Thread.Sleep(3000);
    return string.Format("Hello, {0}", name);
}
```

Чтобы сделать метод такого рода асинхронным, определим метод `GreetingAsync()`. Согласно асинхронному шаблону, основанному на задачах, асинхронный метод имеет имя, заканчивающееся на `Async`, и возвращает задачу. Метод `GreetingAsync()` определен с тем же входным параметром, что и `Greeting()`, но с возвращаемым типом `Task<string>`. Тип `Task<string>` определяет задачу, которая возвращает строку. Проще всего вернуть задачу с применением метода `Task.Run()`. Обобщенная версия метода `Task.Run<string>()` создает задачу, возвращающую строку:

```
static Task<string> GreetingAsync(string name)
{
    return Task.Run<string>(() =>
    {
        return Greeting(name);
    });
}
```

Вызов асинхронного метода

Вызвать асинхронный метод `GreetingAsync()` можно с использованием ключевого слова `await` для возвращенной задачи. Ключевое слово `await` требует, чтобы метод был объявлен с модификатором `async`. Код внутри этого метода не будет продолжать выполнение, пока не завершится метод `GreetingAsync()`. Тем не менее, поток, запустивший метод `CallerWithAsync()`, можно использовать повторно. Этот поток не блокируется:

```
private async static void CallerWithAsync()
{
    string result = await GreetingAsync("Stephanie");
    Console.WriteLine(result);
}
```

Вместо передачи результата из асинхронного метода в переменную можно также применять ключевое слово `await` внутри параметров. В следующем коде результат из метода `GreetingAsync()` ожидается посредством `await`, как и в предыдущем фрагменте кода, но на этот раз результат передается прямо методу `Console.WriteLine()`:

```
private async static void CallerWithAsync2()
{
    Console.WriteLine(await GreetingAsync("Stephanie"));
}
```

На заметку! Модификатор `async` может использоваться только с методами, возвращающими `Task` или `void`. Его нельзя применять для точки входа в программу (для метода `Main()`). Ключевое слово `await` может использоваться только с методами, возвращающими `Task`.

В следующем разделе будет показано, что является движущей силой ключевого слова `await`. На самом деле “за кулисами” применяются задачи продолжения.

Задачи продолжения

Метод `GreetingAsync()` возвращает объект `Task<string>`. Объект `Task` содержит информацию о созданной задаче и позволяет организовать ожидание ее завершения. Метод `ContinueWith()` класса `Task` определяет код, который должен быть вызван, когда задача завершается. Делегат, заданный для метода `ContinueWith()`, принимает завершенную задачу в своем аргументе, что позволяет получить доступ к результату задачи с помощью свойства `Result`:

```
private static void CallerWithContinuationTask()
{
    Task<string> t1 = GreetingAsync("Stephanie");
    t1.ContinueWith(t =>
    {
        string result = t.Result;
        Console.WriteLine(result);
    });
}
```

Компилятор преобразует ключевое слово `await`, помещая весь следующий за ним код внутрь блока метода `ContinueWith()`.

Контекст синхронизации

На протяжении времени существования внутри методов `CallerWithAsync()` и `CallerWithContinuationTask()` используются разные потоки. Один поток применяется для вызова метода `GreetingAsync()`, а другой поток начинает действовать после ключевого слова `await` или внутри блока кода в методе `ContinueWith()`.

В консольном приложении обычно это не является проблемой. Однако необходимо обеспечить, чтобы, по меньшей мере, один поток переднего плана функционировал, прежде чем все фоновые задачи будут завершены. В примере приложения вызывается метод `Console.ReadLine()` для удержания главного потока в режиме выполнения вплоть до нажатия клавиши `<Enter>`.

С другой стороны, в приложениях, которые привязаны к конкретному потоку для выполнения определенных действий (например, в WPF-приложениях элементы пользовательского интерфейса могут быть доступны только из потока пользовательского интерфейса), это будет проблемой.

Применяя ключевые слова `async` и `await`, не потребуется предпринимать никаких специальных действий для доступа к потоку пользовательского интерфейса после завершения `await`. По умолчанию сгенерированный код переключается на поток, имеющий контекст синхронизации. Приложение WPF устанавливает объект `DispatcherSynchronizationContext`, а приложение Windows Forms — объект `WindowsFormsSynchronizationContext`. Если вызываемому потоку асинхронного метода назначен контекст синхронизации, то при продолжении выполнения после `await` по умолчанию используется тот же самый контекст синхронизации. Если тот же самый контекст синхронизации не должен применяться, потребуется вызвать метод `ConfigureAwaitAwait(continueOnCapturedContext: false)` класса `Task`. Примером может служить WPF-приложение, в котором код, следующий за `await`, не обращается к каким-либо элементам пользовательского интерфейса. В этом случае эффективнее избежать переключения контекста синхронизации.

Использование множества асинхронных методов

Внутри асинхронного метода можно вызывать не только какой-то один, но и несколько асинхронных методов. Способ реализации зависит от того, должны ли результаты, возвращаемые одним асинхронным методом, передаваться другому такому методу.

Последовательный вызов асинхронных методов

Для вызова каждого асинхронного метода можно использовать ключевое слово `await`. В случае, когда один метод зависит от результатов выполнения другого метода, это очень удобно. В приведенном ниже коде второй вызов `GreetingAsync()` совершенно не зависит от результата первого вызова `GreetingAsync()`. Следовательно, метод `MultipleAsyncMethods()` мог бы вернуть результат быстрее, если не применять `await` с каждым отдельным методом:

```
private async static void MultipleAsyncMethods()
{
    string s1 = await GreetingAsync("Stephanie");
    string s2 = await GreetingAsync("Matthias");
    Console.WriteLine("Finished both methods.\n " +
        "Result 1: {0}\n Result 2: {1}", s1, s2);
}
```

Использование комбинаторов

Если асинхронные методы не зависят друг от друга, намного быстрее не указывать `await` для каждого метода по отдельности, а вместо этого присвоить результаты работы асинхронных методов переменным типа `Task`.

Метод `GreetingAsync()` возвращает `Task<string>`. Оба метода теперь могут выполняться параллельно. Помогут в этом *комбинаторы*. Комбинатор принимает множество параметров одного типа и возвращает значение того же самого типа. Передаваемые параметры “комбинируются” в один. Комбинаторы `Task` принимают несколько объектов `Task` в качестве параметров и возвращают объект `Task`.

В следующем примере кода вызывается метод комбинатора `Task.WhenAll()`, к которому можно применить `await` для ожидания завершения обеих задач:

```
private async static void MultipleAsyncMethodsWithCombinators1()
{
    Task<string> t1 = GreetingAsync("Stephanie");
    Task<string> t2 = GreetingAsync("Matthias");
    await Task.WhenAll(t1, t2);
    Console.WriteLine("Finished both methods.\n " +
        "Result 1: {0}\n Result 2: {1}", t1.Result, t2.Result);
}
```

В классе `Task` определены комбинаторы `WhenAll()` и `WhenAny()`. Задача, возвращаемая из метода `WhenAll()`, завершается, когда завершены все задачи, переданные методу; задача, возвращаемая из метода `WhenAny()`, завершается, когда завершена одна из задач, переданных методу.

Для метода `WhenAll()` в классе `Task` предусмотрено несколько перегруженных версий. Если все задачи возвращают один и тот же тип, `await` можно применить к массиву этого типа. Метод `GreetingAsync()` возвращает `Task<string>`, и результатом `await` для этого метода будет `string`. Следовательно, метод `Task.WhenAll()` можно использовать для возврата массива `string`:

```
private async static void MultipleAsyncMethodsWithCombinators2()
{
    Task<string> t1 = GreetingAsync("Stephanie");
    Task<string> t2 = GreetingAsync("Matthias");
    string[] result = await Task.WhenAll(t1, t2);
    Console.WriteLine("Finished both methods.\n " +
        "Result 1: {0}\n Result 2: {1}", result[0], result[1]);
}
```

Преобразование асинхронного шаблона

Не все классы в `.NET Framework` поддерживают новый стиль асинхронных методов, появившийся в версии `.NET 4.5`. Многие классы по-прежнему предлагают реализацию асинхронного шаблона с методами `BeginXXX()` и `EndXXX()`, а не асинхронные методы, основанные на задачах.

Для начала давайте создадим асинхронный метод из ранее определенного синхронного метода `Greeting()` с помощью делегата. Метод `Greeting()` получает строку в качестве параметра и возвращает строку, поэтому для ссылки на данный метод применяется переменная типа делегата `Func<string, string>`. Согласно асинхронному шаблону, в дополнение к параметру `AsyncCallback` метод `BeginGreeting()` принимает параметр типа `string` и возвращает `IAsyncResult`. Метод `EndGreeting()` возвращает результат, полученный из метода `Greeting()` — объект `string` — и принимает параметр типа `IAsyncResult`. Для асинхронной реализации используется только упомянутый делегат.

```
private static Func<string, string> greetingInvoker = Greeting;
static IAsyncResult BeginGreeting(string name, AsyncCallback callback,
    object state)
{
    return greetingInvoker.BeginInvoke(name, callback, state);
}
```

```

static string EndGreeting(IAsyncResult ar)
{
    return greetingInvoker.EndInvoke(ar);
}

```

Теперь методы `BeginGreeting()` и `EndGreeting()` доступны, и можно приступить к преобразованию кода с целью применения ключевых слов `async` и `await` для получения результата. В классе `TaskFactory` определен метод `FromAsync()`, который позволяет преобразовать методы, использующие асинхронный шаблон, в шаблон TAP.

В приведенном ниже примере кода первый обобщенный параметр типа `Task`, `Task<string>`, определяет возвращаемое значение вызываемого метода. Обобщенный параметр метода `FromAsync()` определяет входной тип метода. В этом случае входным типом снова является `string`. Первые два параметра метода `FromAsync()` имеют типы делегатов для передачи адресов методов `BeginGreeting()` и `EndGreeting()`. За ними следуют входные параметры и параметр состояния объекта. Состояние объекта не используется, так что для него указывается `null`. Поскольку метод `FromAsync()` возвращает тип `Task`, в примере кода `Task<string>` ключевое слово `await` может применяться следующим образом:

```

private static async void ConvertingAsyncPattern()
{
    string s = await Task<string>.Factory.FromAsync<string>(
        BeginGreeting, EndGreeting, "Angela", null);
    Console.WriteLine(s);
}

```

Обработка ошибок

Подробное описание обработки ошибок и исключений представлено в главе 16. Тем не менее, в контексте асинхронных методов вы должны быть осведомлены о некоторых специальных способах обработки ошибок. Начнем с простого метода, который генерирует исключение после заданной паузы (файл `ErrorHandling/Program.cs`):

```

static async Task ThrowAfter(int ms, string message)
{
    await Task.Delay(ms);
    throw new Exception(message);
}

```

Если асинхронный метод вызывается без `await`, вызов можно поместить внутрь блока `try/catch`, однако исключение не будет перехвачено. Причина в том, что метод `DontHandle()` завершится до того, как сгенерируется исключение в `ThrowAfter()`. К вызову метода `ThrowAfter()` понадобится применить ключевое слово `await`.

```

private static void DontHandle()
{
    try
    {
        ThrowAfter(200, "first");
        // Исключение не будет перехвачено, т.к. этот метод
        // завершится до его генерации.
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

```

Внимание! К асинхронным методам, возвращающим `void`, ключевое слово `await` не может быть применено. Проблема в том, что исключения, генерируемые внутри методов `async void`, не могут быть перехвачены. Именно поэтому лучше возвращать из асинхронного метода тип `Task`. Данное правило не распространяется на методы обработчиков или переопределенные методы базовых классов.

Обработка исключений, возникающих в асинхронных методах

Удобный способ обработки исключений, возникающих в асинхронных методах, заключается в применении ключевого слова `await` и помещении вызовов этих методов внутрь оператора `try/catch`, как показано в приведенном фрагменте кода. Метод `HandleOneError()` освобождает поток после вызова метода `ThrowAfter()` асинхронным образом, но сохраняет объект `Task` для продолжения работы после того, как задача завершается. Когда это происходит (в данном случае, когда исключение генерируется через две секунды), условие в `catch` дает совпадение и код внутри блока `catch` выполняется:

```
private static async void HandleOneError()
{
    try
    {
        await ThrowAfter(2000, "first");
    }
    catch (Exception ex)
    {
        Console.WriteLine("handled {0}", ex.Message);
    }
}
```

Исключения, возникающие в нескольких асинхронных методах

Что делать в ситуации, когда оба асинхронных метода, которые были вызваны, генерируют исключения? В следующем примере вызывается первый метод `ThrowAfter()`, который через две секунды генерирует исключение с сообщением "first". По его завершении этого вызова метод `ThrowAfter()` вызывается еще раз для того, чтобы спустя одну секунду сгенерировать исключение с сообщением "second". Так как первый вызов метода `ThrowAfter()` уже сгенерировал исключение, код внутри блока `try` выполняться больше не будет и до второго вызова дело не дойдет, а управление будет передано в блок `catch` для обработки первого исключения.

```
private static async void StartTwoTasks()
{
    try
    {
        await ThrowAfter(2000, "first");
        await ThrowAfter(1000, "second"); // Второй вызов не будет выполнен,
        // поскольку первый вызов метода генерирует исключение.
    }
    catch (Exception ex)
    {
        Console.WriteLine("handled {0}", ex.Message);
    }
}
```

Теперь давайте обеспечим параллельную работу двух вызовов `ThrowAfter()`. Первый вызов генерирует исключение через две секунды, а второй — через одну секунду. За счет применения метода `Task.WhenAll()` организуется ожидание завершения обеих задач, независимо от того, возникло исключение или нет. Следовательно, спустя две секунды вызов

`Task.WhenAll()` завершится, а исключение будет перехвачено оператором `catch`. Однако вы увидите только информацию об исключении из первого вызова `ThrowAfter()`, которая была передана методу `WhenAll()`. Это не та задача, которая сгенерировала исключение первой (на самом деле это сделала вторая задача), но это первая задача в списке.

```
private async static void StartTwoTasksParallel()
{
    try
    {
        Task t1 = ThrowAfter(2000, "first");
        Task t2 = ThrowAfter(1000, "second");
        await Task.WhenAll(t1, t2);
    }
    catch (Exception ex)
    {
        // Просто отображает информацию об исключении, возникшем
        // в первой задаче, ожидаемой с помощью метода WhenAll()
        Console.WriteLine("handled {0}", ex.Message);
    }
}
```

Один из способов получения информации об исключениях из всех задач предусматривает объявление переменных `t1` и `t2` для представления задач за пределами блока `try`, что позволит обращаться к ним в блоке `catch`. Здесь с помощью свойства `IsFaulted` можно проверить состояние задачи для определения состояния сбоя. Если было сгенерировано исключение, свойство `IsFaulted` возвратит `true`. Информацию об исключении можно получить с использованием свойства `Exception.InnerException` класса `Task`. Другой, обычно более эффективный способ извлечения сведений об исключениях из всех задач демонстрируется в следующем разделе.

Использование информации типа `AggregateException`

Чтобы получить сведения об исключении из всех потерпевших отказ задач, результат выполнения метода `Task.WhenAll()` может быть сохранен в переменной типа `Task`. Затем к этой переменной применяется `await` для ожидания до тех пор, пока все задачи не будут завершены. В противном случае исключение по-прежнему будет утеряно. Как будет показано в последнем разделе, с помощью оператора `catch` может быть перехвачено только исключение, возникшее в первой задаче. Однако теперь имеется доступ к свойству `Exception` внешней задачи. Свойство `Exception` имеет тип `AggregateException`. В данном типе определено свойство `InnerExceptions` (помимо `InnerException`), которое содержит список всех исключений, сгенерированных в ожидаемых задачах. Это позволяет легко проходить по всем исключениям:

```
private static async void ShowAggregatedException()
{
    Task taskResult = null;
    try
    {
        Task t1 = ThrowAfter(2000, "first");
        Task t2 = ThrowAfter(1000, "second");
        await (taskResult = Task.WhenAll(t1, t2));
    }
    catch (Exception ex)
    {
        Console.WriteLine("handled {0}", ex.Message);
        foreach (var ex1 in taskResult.Exception.InnerExceptions)
        {
            Console.WriteLine("inner exception {0}", ex1.Message);
        }
    }
}
```



```

cts = new CancellationTokenSource();
try
{
    foreach (var req in GetSearchRequests())
    {
        var client = new HttpClient();
        var response = await client.GetAsync(req.Url, cts.Token);
        string resp = await response.Content.ReadAsStringAsync();
        //...
    }
}
catch (OperationCanceledException ex)
{
    MessageBox.Show(ex.Message);
}
}

```

Отмена выполнения специальных задач

А как быть с отменой выполнения специальных задач? Метод `Run()` класса `Task` предлагает перегруженную версию, которая также принимает `CancellationToken`. Тем не менее, специальные задачи должны проверять, была ли запрошена отмена. В показанном ниже примере это реализовано внутри цикла `foreach`. Маркер может быть проверен с помощью свойства `IsCancellationRequested`. Если перед генерацией исключения требуется определенная очистка, то лучше предварительно проверить, действительно ли запрошена отмена. Если же очистка не нужна, исключение может быть сгенерировано немедленно после проверки, что и делается с помощью метода `ThrowIfCancellationRequested()`.

```

await Task.Run(() =>
{
    var images = req.Parse(resp);
    foreach (var image in images)
    {
        cts.Token.ThrowIfCancellationRequested();
        searchInfo.List.Add(image);
    }
}, cts.Token);

```

Таким образом, пользователи получили возможность отменять длительно выполняющиеся задачи.

Резюме

В этой главе были представлены ключевые слова `async` и `await`, появившиеся в версии C# 5. На нескольких примерах были продемонстрированы преимущества асинхронного шаблона, основанного на задачах, по сравнению с обычным асинхронным шаблоном и асинхронным шаблоном, основанным на событиях, которые доступны в более ранних версиях .NET.

Вы также увидели, насколько легко создавать асинхронные методы с помощью класса `Task`, и узнали, как использовать ключевые слова `async` и `await` для ожидания завершения этих методов, не блокируя потоки. Наконец, вы ознакомились с особенностями обработки ошибок в асинхронных методах.

За дополнительными сведениями о параллельном программировании, потоках и задачах обращайтесь в главу 21.

В следующей главе мы продолжим рассматривать основные средства C# и .NET, обратившись к вопросам управления памятью и ресурсами.