

# Содержание

Об авторах	34
Благодарности	35
<b>Введение</b>	<b>36</b>
Авторы и читатели — одна команда	36
Краткий обзор книги	36
Часть I. Язык программирования C# и платформа .NET 5	37
Часть II. Основы программирования на C#	37
Часть III. Объектно-ориентированное программирование на C#	38
Часть IV. Дополнительные конструкции программирования на C#	39
Часть V. Программирование с использованием сборок .NET Core	40
Часть VI. Работа с файлами, сериализация объектов и доступ к данным	41
Часть VII. Entity Framework Core	42
Часть VIII. Разработка клиентских приложений для Windows	42
Часть IX. ASP.NET Core	44
Ждем ваших отзывов!	45
<b>Часть I. Язык программирования C# и платформа .NET 5</b>	<b>47</b>
<b>Глава 1. Введение в C# и .NET (Core) 5</b>	<b>48</b>
Некоторые основные преимущества инфраструктуры .NET Core	49
Понятие жизненного цикла поддержки .NET Core	50
Предварительный обзор строительных блоков .NET Core (.NET Runtime, CTS и CLS)	51
Роль библиотек базовых классов	52
Роль .NET Standard	52
Что привносит язык C#	52
Основные средства в предшествующих выпусках	53
Новые средства в C# 9	56
Сравнение управляемого и неуправляемого кода	57
Использование дополнительных языков программирования, ориентированных на .NET Core	57
Обзор сборок .NET	57
Роль языка CIL	58
Преимущества языка CIL	61
Роль метаданных типов .NET Core	62
Роль манифеста сборки	63
Понятие общей системы типов	64
Типы классов CTS	64
Типы интерфейсов CTS	64
Типы структур CTS	65
Типы перечислений CTS	66
Типы делегатов CTS	66
Члены типов CTS	67
Встроенные типы данных CTS	67
Понятие общезыковой спецификации	68
Обеспечение совместимости с CLS	70

Понятие .NET Core Runtime	70
Различия между сборкой, пространством имен и типом	70
Доступ к пространству имен программным образом	71
Ссылка на внешние сборки	73
Исследование сборки с помощью <code>ildasm.exe</code>	74
Резюме	75
<b>Глава 2. Создание приложений на языке C#</b>	76
Установка .NET 5	76
Понятие схемы нумерации версий .NET 5	77
Подтверждение успешности установки .NET 5	77
Использование более ранних версий .NET (Core) SDK	78
Построение приложений .NET Core с помощью Visual Studio	78
Установка Visual Studio 2019 (Windows)	79
Испытание Visual Studio 2019	80
Построение приложений .NET Core с помощью Visual Studio Code	90
Испытание Visual Studio Code	90
Документация по .NET Core и C#	93
Резюме	94
<b>Часть II. Основы программирования на C#</b>	95
<b>Глава 3. Главные конструкции программирования на C#: часть 1</b>	96
Структура простой программы C#	96
Использование вариаций метода <code>Main()</code> (обновление в версии 7.1)	98
Использование операторов верхнего уровня (нововведение в версии 9.0)	99
Указание кода ошибки приложения (обновление в версии 9.0)	101
Обработка аргументов командной строки	103
Указание аргументов командной строки в Visual Studio	105
Интересное отступление от темы: некоторые дополнительные члены	
класса <code>System.Environment</code>	105
Использование класса <code>System.Console</code>	107
Выполнение базового ввода и вывода с помощью класса <code>Console</code>	107
Форматирование консольного вывода	108
Форматирование числовых данных	109
Форматирование числовых данных за рамками консольных приложений	111
Работа с системными типами данных и соответствующими ключевыми	
словами C#	111
Объявление и инициализация переменных	112
Использование внутренних типов данных и операции <code>new</code>	
(обновление в версии 9.0)	114
Иерархия классов для типов данных	115
Члены числовых типов данных	117
Члены <code>System.Boolean</code>	117
Члены <code>System.Char</code>	118
Разбор значений из строковых данных	118
Использование метода <code>TryParse()</code> для разбора значений	
из строковых данных	119

Использование типов <code>System.DateTime</code> и <code>System.TimeSpan</code>	120
Работа с пространством имен <code>System.Numerics</code>	120
Использование разделителей групп цифр (нововведение в версии 7.0)	122
Использование двоичных литералов (нововведение в версии 7.0/7.2)	122
Работа со строковыми данными	123
Выполнение базовых манипуляций со строками	123
Выполнение конкатенации строк	124
Использование управляющих последовательностей	125
Выполнение интерполяции строк	126
Определение дословных строк (обновление в версии 8.0)	127
Работа со строками и операциями равенства	128
Строки неизменяемы	130
Использование типа <code>System.Text.StringBuilder</code>	132
Сужающие и расширяющие преобразования типов данных	133
Использование ключевого слова <code>checked</code>	135
Настройка проверки переполнения на уровне проекта	137
Настройка проверки переполнения на уровне проекта (Visual Studio)	137
Использование ключевого слова <code>unchecked</code>	138
Неявно типизированные локальные переменные	138
Неявное объявление чисел	140
Ограничения неявно типизированных переменных	140
Неявно типизированные данные строго типизированы	141
Полезность неявно типизированных локальных переменных	142
Работа с итерационными конструкциями <code>C#</code>	143
Использование цикла <code>for</code>	143
Использование цикла <code>foreach</code>	144
Использование неявной типизации в конструкциях <code>foreach</code>	144
Использование циклов <code>while</code> и <code>do/while</code>	145
Краткое обсуждение области видимости	146
Работа с конструкциями принятия решений и операциями отношения/равенства	146
Использование оператора <code>if/else</code>	147
Использование операций отношения и равенства	147
Использование операторов <code>if/else</code> и сопоставления с образцом (нововведение в версии 7.0)	148
Внесение улучшений в сопоставление с образцом (нововведение в версии 9.0)	149
Использование условной операции (обновление в версиях 7.2, 9.0)	150
Использование логических операций	151
Использование оператора <code>switch</code>	152
Выполнение сопоставления с образцом в операторах <code>switch</code> (нововведение в версии 7.0, обновление в версии 9.0)	155
Использование выражений <code>switch</code> (нововведение в версии 8.0)	158
Резюме	159
<b>Глава 4. Главные конструкции программирования на <code>C#</code>: часть 2</b>	160
Понятие массивов <code>C#</code>	160
Синтаксис инициализации массивов <code>C#</code>	161
Понятие неявно типизированных локальных массивов	162

Определение массива объектов	163
Работа с многомерными массивами	164
Использование массивов в качестве аргументов и возвращаемых значений	165
Использование базового класса <code>System.Array</code>	166
Использование индексов и диапазонов (нововведение в версии 8.0)	167
Понятие методов	169
Члены, сжатые до выражений	169
Локальные функции (нововведение в версии 7.0, обновление в версии 9.0)	170
Статические локальные функции (нововведение в версии 8.0)	171
Понятие параметров методов	172
Модификаторы параметров для методов	172
Стандартное поведение передачи параметров	173
Использование модификатора <code>out</code> (обновление в версии 7.0)	174
Использование модификатора <code>ref</code>	176
Использование модификатора <code>in</code> (нововведение в версии 7.2)	177
Использование модификатора <code>params</code>	178
Определение необязательных параметров	180
Использование именованных параметров (обновление в версии 7.2)	181
Понятие перегрузки методов	182
Понятие типа <code>enum</code>	185
Управление хранилищем, лежащим в основе перечисления	186
Объявление переменных типа перечисления	187
Использование типа <code>System.Enum</code>	188
Динамическое обнаружение пар “имя-значение” перечисления	188
Использование перечислений, флагов и побитовых операций	190
Понятие структуры (как типа значения)	192
Создание переменных типа структур	193
Использование структур, допускающих только чтение (нововведение в версии 7.2)	194
Использование членов, допускающих только чтение (нововведение в версии 8.0)	195
Использование структур <code>ref</code> (нововведение в версии 7.2)	195
Использование освобождаемых структур <code>ref</code> (нововведение в версии 8.0)	196
Типы значений и ссылочные типы	197
Использование типов значений, ссылочных типов и операции присваивания	198
Использование типов значений, содержащих ссылочные типы	199
Передача ссылочных типов по значению	201
Передача ссылочных типов по ссылке	202
Заключительные детали относительно типов значений и ссылочных типов	203
Понятие типов <code>C#</code> , допускающих <code>null</code>	204
Использование типов значений, допускающих <code>null</code>	205
Использование ссылочных типов, допускающих <code>null</code> (нововведение в версии 8.0)	207
Работа с типами, допускающими значение <code>null</code>	209
Понятие кортежей (нововведение и обновление в версии 7.0)	212
Начало работы с кортежами	212
Использование выведенных имен переменных (обновление в версии <code>C# 7.1</code> )	213
Понятие эквивалентности/неэквивалентности кортежей (нововведение в версии 7.3)	214

Использование кортежей как возвращаемых значений методов	214
Использование отбрасывания с кортежами	215
Использование выражений <code>switch</code> с сопоставлением с образцом для кортежей (нововведение в версии 8.0)	215
Деконструирование кортежей	216
Резюме	217
<b>Часть III. Объектно-ориентированное программирование на C#</b>	219
<b>Глава 5. Инкапсуляция</b>	220
Знакомство с типом класса C#	220
Размещение объектов с помощью ключевого слова <code>new</code>	222
Понятие конструкторов	223
Роль стандартного конструктора	223
Определение специальных конструкторов	224
Еще раз о стандартном конструкторе	226
Роль ключевого слова <code>this</code>	227
Построение цепочки вызовов конструкторов с использованием <code>this</code>	228
Исследование потока управления конструкторов	231
Еще раз о необязательных аргументах	232
Понятие ключевого слова <code>static</code>	233
Определение статических полей данных	234
Определение статических методов	236
Определение статических конструкторов	237
Определение статических классов	240
Импортирование статических членов с применением ключевого слова <code>using</code> языка C#	241
Основные принципы объектно-ориентированного программирования	242
Роль инкапсуляции	242
Роль наследования	243
Роль полиморфизма	244
Модификаторы доступа C# (обновление в версии 7.2)	245
Использование стандартных модификаторов доступа	246
Использование модификаторов доступа и вложенных типов	247
Первый принцип объектно-ориентированного программирования: службы инкапсуляции C#	248
Инкапсуляция с использованием традиционных методов доступа и изменения	249
Инкапсуляция с использованием свойств	251
Использование свойств внутри определения класса	254
Свойства, допускающие только чтение	256
Свойства, допускающие только запись	257
Смешивание закрытых и открытых методов <code>get/set</code> в свойствах	257
Еще раз о ключевом слове <code>static</code> : определение статических свойств	257
Сопоставление с образцом и шаблоны свойств (нововведение в версии 8.0)	258
Понятие автоматических свойств	259
Взаимодействие с автоматическими свойствами	261
Автоматические свойства и стандартные значения	261
Инициализация автоматических свойств	263

Понятие инициализации объектов	264
Обзор синтаксиса инициализации объектов	264
Использование средства доступа только для инициализации (нововведение в версии 9.0)	265
Вызов специальных конструкторов с помощью синтаксиса инициализации	266
Инициализация данных с помощью синтаксиса инициализации	268
Работа с константными полями данных и полями данных, допускающими только чтение	269
Понятие константных полей данных	269
Понятие полей данных, допускающих только чтение	270
Понятие статических полей, допускающих только чтение	271
Понятие частных классов	272
Использование записей (нововведение в версии 9.0)	273
Эквивалентность с типами записей	275
Копирование типов записей с использованием выражений <code>with</code>	276
Резюме	277
<b>Глава 6. Наследование и полиморфизм</b>	<b>278</b>
Базовый механизм наследования	278
Указание родительского класса для существующего класса	279
Замечание относительно множества базовых классов	281
Использование ключевого слова <code>sealed</code>	281
Еще раз о диаграммах классов Visual Studio	282
Второй принцип объектно-ориентированного программирования: детали наследования	284
Вызов конструкторов базового класса с помощью ключевого слова <code>base</code>	285
Хранение секретов семейства: ключевое слово <code>protected</code>	287
Добавление запечатанного класса	288
Наследование с типами записей (нововведение в версии 9.0)	289
Реализация модели включения/делегации	292
Определения вложенных типов	293
Третий принцип объектно-ориентированного программирования: поддержка полиморфизма в C#	295
Использование ключевых слов <code>virtual</code> и <code>override</code>	296
Переопределение виртуальных членов с помощью Visual Studio/Visual Studio Code	298
Запечатывание виртуальных членов	299
Абстрактные классы	299
Полиморфные интерфейсы	301
Сокрытие членов	304
Правила приведения для базовых и производных классов	306
Использование ключевого слова <code>as</code>	308
Использование ключевого слова <code>is</code> (обновление в версиях 7.0, 9.0)	309
Еще раз о сопоставлении с образцом (нововведение в версии 7.0)	311
Главный родительский класс: <code>System.Object</code>	313
Переопределение метода <code>System.Object.ToString()</code>	316
Переопределение метода <code>System.Object.Equals()</code>	316
Переопределение метода <code>System.Object.GetHashCode()</code>	317

## 12 Содержание

Тестирование модифицированного класса <code>Person</code>	318
Использование статических членов класса <code>System.Object</code>	319
Резюме	320
<b>Глава 7. Структурированная обработка исключений</b>	<b>321</b>
Ода ошибкам, дефектам и исключениям	321
Роль обработки исключений .NET	322
Строительные блоки обработки исключений в .NET	323
Базовый класс <code>System.Exception</code>	324
Простейший пример	325
Генерация общего исключения	327
Перехват исключений	329
Выражение <code>throw</code> (нововведение в версии 7.0)	330
Конфигурирование состояния исключения	330
Свойство <code>TargetSite</code>	331
Свойство <code>StackTrace</code>	331
Свойство <code>HelpLink</code>	332
Свойство <code>Data</code>	333
Исключения уровня системы ( <code>System.SystemException</code> )	335
Исключения уровня приложения ( <code>System.ApplicationException</code> )	335
Построение специальных исключений, способ первый	336
Построение специальных исключений, способ второй	338
Построение специальных исключений, способ третий	338
Обработка множества исключений	340
Общие операторы <code>catch</code>	342
Повторная генерация исключений	343
Внутренние исключения	343
Блок <code>finally</code>	344
Фильтры исключений	345
Отладка необработанных исключений с использованием Visual Studio	346
Резюме	347
<b>Глава 8. Работа с интерфейсами</b>	<b>348</b>
Понятие интерфейсных типов	348
Сравнение интерфейсных типов и абстрактных базовых классов	349
Определение специальных интерфейсов	352
Реализация интерфейса	353
Обращение к членам интерфейса на уровне объектов	356
Получение ссылок на интерфейсы: ключевое слово <code>as</code>	357
Получение ссылок на интерфейсы: ключевое слово <code>is</code> (обновление в версии 7.0)	357
Стандартные реализации (нововведение в версии 8.0)	357
Статические конструкторы и члены (нововведение в версии 8.0)	359
Использование интерфейсов в качестве параметров	359
Использование интерфейсов в качестве возвращаемых значений	361
Массивы интерфейсных типов	362
Автоматическая реализация интерфейсов	363
Явная реализация интерфейсов	365

Проектирование иерархий интерфейсов	367
Иерархии интерфейсов со стандартными реализациями (нововведение в версии 8.0)	369
Множественное наследование с помощью интерфейсных типов	370
Интерфейсы <code>IEnumerable</code> и <code>IEnumerator</code>	373
Построение итераторных методов с использованием ключевого слова <code>yield</code>	375
Построение именованного итератора	378
Интерфейс <code>ICloneable</code>	379
Более сложный пример клонирования	381
Интерфейс <code>IComparable</code>	383
Указание множества порядков сортировки с помощью <code>IComparer</code>	386
Специальные свойства и специальные типы сортировки	388
Резюме	388
<b>Глава 9. Время существования объектов</b>	<b>389</b>
Классы, объекты и ссылки	389
Базовые сведения о времени жизни объектов	391
Код CIL для ключевого слова <code>new</code>	391
Установка объектных ссылок в <code>null</code>	393
Выяснение, нужен ли объект	394
Понятие поколений объектов	395
Эфемерные поколения и сегменты	397
Типы сборки мусора	397
Фоновая сборка мусора	398
Тип <code>System.GC</code>	398
Принудительный запуск сборщика мусора	400
Построение финализируемых объектов	402
Переопределение метода <code>System.Object.Finalize()</code>	403
Подробности процесса финализации	405
Построение освобождаемых объектов	406
Повторное использование ключевого слова <code>using</code> в C#	408
Объявления <code>using</code> (нововведение в версии 8.0)	409
Создание финализируемых и освобождаемых типов	410
Формализованный шаблон освобождения	411
Ленивое создание объектов	413
Настройка процесса создания данных <code>Lazy&lt;&gt;</code>	416
Резюме	417
<b>Часть IV. Дополнительные конструкции программирования на C#</b>	<b>419</b>
<b>Глава 10. Коллекции и обобщения</b>	<b>420</b>
Побудительные причины создания классов коллекций	420
Пространство имен <code>System.Collections</code>	422
Обзор пространства имен <code>System.Collections.Specialized</code>	424
Проблемы, присущие необобщенным коллекциям	425
Проблема производительности	425
Проблема безопасности в отношении типов	429
Первый взгляд на обобщенные коллекции	432

Роль параметров обобщенных типов	433
Указание параметров типа для обобщенных классов и структур	434
Указание параметров типа для обобщенных членов	436
Указание параметров типов для обобщенных интерфейсов	436
Пространство имен <code>System.Collections.Generic</code>	437
Синтаксис инициализации коллекций	439
Работа с классом <code>List&lt;T&gt;</code>	440
Работа с классом <code>Stack&lt;T&gt;</code>	442
Работа с классом <code>Queue&lt;T&gt;</code>	443
Работа с классом <code>SortedSet&lt;T&gt;</code>	444
Работа с классом <code>Dictionary&lt;TKey, TValue&gt;</code>	446
Пространство имен <code>System.Collections.ObjectModel</code>	447
Работа с классом <code>ObservableCollection&lt;T&gt;</code>	448
Создание специальных обобщенных методов	450
Выведение параметров типа	452
Создание специальных обобщенных структур и классов	452
Выражения <code>default</code> вида значений в обобщениях	454
Выражения <code>default</code> литерального вида (нововведение в версии 7.1)	455
Сопоставление с образцом в обобщениях (нововведение в версии 7.1)	455
Ограничение параметров типа	456
Примеры использования ключевого слова <code>where</code>	456
Отсутствие ограничений операций	458
Резюме	459
<b>Глава 11. Расширенные средства языка C#</b>	460
Понятие индексаторных методов	460
Индексация данных с использованием строковых значений	462
Перегрузка индексаторных методов	464
Многомерные индексаторы	464
Определения индексаторов в интерфейсных типах	465
Понятие перегрузки операций	466
Перегрузка бинарных операций	467
А как насчет операций <code>+=</code> и <code>-=</code> ?	469
Перегрузка унарных операций	469
Перегрузка операций эквивалентности	470
Перегрузка операций сравнения	471
Финальные соображения относительно перегрузки операций	471
Понятие специальных преобразований типов	472
Повторение: числовые преобразования	472
Повторение: преобразования между связанными типами классов	472
Создание специальных процедур преобразования	473
Дополнительные явные преобразования для типа <code>Square</code>	476
Определение процедур неявного преобразования	477
Понятие расширяющих методов	478
Определение расширяющих методов	478
Вызов расширяющих методов	480
Импортирование расширяющих методов	480
Расширение типов, реализующих специфичные интерфейсы	481
Поддержка расширяющего метода <code>GetEnumerator()</code> (нововведение в версии 9.0)	482

Понятие анонимных типов	484
Определение анонимного типа	484
Внутреннее представление анонимных типов	485
Реализация методов ToString() и GetHashCode()	487
Семантика эквивалентности анонимных типов	487
Анонимные типы, содержащие другие анонимные типы	489
Работа с типами указателей	490
Ключевое слово unsafe	492
Работа с операциями * и &	493
Небезопасная (и безопасная) функция обмена	494
Доступ к полям через указатели (операция ->)	495
Ключевое слово stackalloc	495
Закрепление типа посредством ключевого слова fixed	496
Ключевое слово sizeof	497
Резюме	497
<b>Глава 12. Делегаты, события и лямбда-выражения</b>	<b>499</b>
Понятие типа делегата	500
Определение типа делегата в C#	500
Базовые классы System.MulticastDelegate и System.Delegate	503
Пример простейшего делегата	504
Исследование объекта делегата	506
Отправка уведомлений о состоянии объекта с использованием делегатов	507
Включение группового вызова	510
Удаление целей из списка вызовов делегата	511
Синтаксис групповых преобразований методов	512
Понятие обобщенных делегатов	513
Обобщенные делегаты Action<> и Func<>	514
Понятие событий C#	516
Ключевое слово event	518
“За кулисами” событий	519
Прослушивание входящих событий	521
Упрощение регистрации событий с использованием Visual Studio	522
Создание специальных аргументов событий	523
Обобщенный делегат EventHandler<T>	525
Понятие анонимных методов C#	525
Доступ к локальным переменным	527
Использование ключевого слова static с анонимными методами (нововведение в версии 9.0)	528
Использование отбрасывания с анонимными методами (нововведение в версии 9.0)	529
Понятие лямбда-выражений	529
Анализ лямбда-выражения	532
Обработка аргументов внутри множества операторов	533
Лямбда-выражения с несколькими параметрами и без параметров	534
Использование ключевого слова static с лямбда-выражениями (нововведение в версии 9.0)	535
Использование отбрасывания с лямбда-выражениями (нововведение в версии 9.0)	536

Модернизация примера CarEvents с использованием лямбда-выражений	536
Лямбда-выражения и члены, сжатые до выражений (обновление в версии 7.0)	537
Резюме	538
<b>Глава 13. LINQ to Objects</b>	<b>539</b>
Программные конструкции, специфичные для LINQ	539
Неявная типизация локальных переменных	540
Синтаксис инициализации объектов и коллекций	541
Лямбда-выражения	541
Расширяющие методы	542
Анонимные типы	543
Роль LINQ	543
Выражения LINQ строго типизированы	545
Основные сборки LINQ	545
Применение запросов LINQ к элементарным массивам	545
Решение с использованием расширяющих методов	546
Решение без использования LINQ	547
Выполнение рефлексии результирующего набора LINQ	548
LINQ и неявно типизированные локальные переменные	549
LINQ и расширяющие методы	551
Роль отложенного выполнения	551
Роль немедленного выполнения	553
Возвращение результатов запроса LINQ	554
Возвращение результатов LINQ посредством немедленного выполнения	555
Применение запросов LINQ к объектам коллекций	556
Доступ к содержащимся в контейнере подобъектам	557
Применение запросов LINQ к необобщенным коллекциям	557
Фильтрация данных с использованием метода OfType<T>()	558
Исследование операций запросов LINQ	559
Базовый синтаксис выборки	560
Получение подмножества данных	561
Проецирование в новые типы данных	562
Проецирование в другие типы данных	563
Подсчет количества с использованием класса Enumerable	564
Изменение порядка следования элементов в результирующих наборах на противоположный	564
Выражения сортировки	564
LINQ как лучшее средство построения диаграмм Венна	565
Устранение дубликатов	566
Операции агрегирования LINQ	567
Внутреннее представление операторов запросов LINQ	567
Построение выражений запросов с применением операций запросов	568
Построение выражений запросов с использованием типа Enumerable и лямбда-выражений	569
Построение выражений запросов с использованием типа Enumerable и анонимных методов	570
Построение выражений запросов с использованием типа Enumerable и низкоуровневых делегатов	571
Резюме	572

<b>Глава 14. Процессы, домены приложений и контексты загрузки</b>	574
Роль процесса Windows	574
Роль потоков	575
Взаимодействие с процессами, используя платформу .NET Core	577
Перечисление выполняющихся процессов	579
Исследование конкретного процесса	580
Исследование набора потоков процесса	580
Исследование набора модулей процесса	582
Запуск и останов процессов программным образом	583
Управление запуском процесса с использованием класса <code>ProcessStartInfo</code>	585
Использование команд операционной системы с классом <code>ProcessStartInfo</code>	586
Домены приложений .NET	587
Класс <code>System.AppDomain</code>	587
Взаимодействие со стандартным доменом приложения	588
Перечисление загруженных сборок	589
Изоляция сборки с помощью контекстов загрузки приложений	590
Итоговые сведения о процессах, доменах приложений и контекстах загрузки	593
Резюме	593
<b>Глава 15. Многопоточное, параллельное и асинхронное программирование</b>	594
Отношения между процессом, доменом приложения, контекстом и потоком	595
Сложность, связанная с параллелизмом	596
Роль синхронизации потоков	596
Пространство имен <code>System.Threading</code>	597
Класс <code>System.Threading.Thread</code>	598
Получение статистических данных о текущем потоке выполнения	599
Свойство <code>Name</code>	599
Свойство <code>Priority</code>	600
Ручное создание вторичных потоков	600
Работа с делегатом <code>ThreadStart</code>	601
Работа с делегатом <code>ParametrizedThreadStart</code>	603
Класс <code>AutoResetEvent</code>	604
Потоки переднего плана и фоновые потоки	605
Проблема параллелизма	606
Синхронизация с использованием ключевого слова <code>lock</code> языка C#	608
Синхронизация с использованием типа <code>System.Threading.Monitor</code>	610
Синхронизация с использованием типа <code>System.Threading.Interlocked</code>	611
Программирование с использованием обратных вызовов <code>Timer</code>	612
Использование автономного отбрасывания (нововведение в версии 7.0)	613
Класс <code>ThreadPool</code>	614
Параллельное программирование с использованием TPL	615
Пространство имен <code>System.Threading.Tasks</code>	616
Роль класса <code>Parallel</code>	616
Обеспечение параллелизма данных с помощью класса <code>Parallel</code>	616
Доступ к элементам пользовательского интерфейса во вторичных потоках	620
Класс <code>Task</code>	621
Обработка запроса на отмену	622

Обеспечение параллелизма задач с помощью класса <code>Parallel</code>	623
Запросы <code>Parallel LINQ (PLINQ)</code>	626
Создание запроса <code>PLINQ</code>	628
Отмена запроса <code>PLINQ</code>	628
Асинхронные вызовы с помощью <code>async/await</code>	629
Знакомство с ключевыми словами <code>async</code> и <code>await</code> языка C# (обновление в версиях 7.1, 9.0)	630
Класс <code>SynchronizationContext</code> и <code>async/await</code>	631
Роль метода <code>ConfigureAwait()</code>	632
Соглашения об именовании асинхронных методов	633
Асинхронные методы, возвращающие <code>void</code>	633
Асинхронные методы с множеством контекстов <code>await</code>	634
Вызов асинхронных методов из неасинхронных методов	636
Ожидание с помощью <code>await</code> в блоках <code>catch</code> и <code>finally</code>	636
Обобщенные возвращаемые типы в асинхронных методах (нововведение в версии 7.0)	637
Локальные функции (нововведение в версии 7.0)	637
Отмена операций <code>async/await</code>	638
Асинхронные потоки (нововведение в версии 8.0)	641
Итоговые сведения о ключевых словах <code>async</code> и <code>await</code>	641
Резюме	642
<b>Часть V. Программирование с использованием сборок .NET Core</b>	643
<b>Глава 16. Построение и конфигурирование библиотек классов</b>	644
Определение специальных пространств имен	644
Разрешение конфликтов имен с помощью полностью заданных имен	646
Разрешение конфликтов имен с помощью псевдонимов	647
Создание вложенных пространств имен	648
Изменение стандартного пространства имен в Visual Studio	649
Роль сборки .NET Core	650
Сборки содействуют многократному использованию кода	650
Сборки устанавливают границы типов	651
Сборки являются единицами, поддерживающими версии	651
Сборки являются самоописательными	651
Формат сборки .NET Core	652
Установка инструментов профилирования C++	652
Заголовок файла операционной системы (Windows)	652
Заголовок файла CLR	654
Код CIL, метаданные типов и манифест сборки	655
Дополнительные ресурсы сборки	655
Отличия между библиотеками классов и консольными приложениями	656
Отличия между библиотеками классов .NET Standard и .NET Core	656
Конфигурирование приложений	657
Построение и потребление библиотеки классов .NET Core	659
Исследование манифеста	661
Исследование кода CIL	663
Исследование метаданных типов	663

Построение клиентского приложения C#	664
Построение клиентского приложения Visual Basic	666
Межязыковое наследование в действии	667
Открытие доступа к внутренним типам для других сборок	668
NuGet и .NET Core	669
Пакетирование сборок с помощью NuGet	669
Ссылка на пакеты NuGet	670
Опубликование консольных приложений (обновление в версии .NET 5)	672
Опубликование приложений, зависящих от инфраструктуры	672
Опубликование автономных приложений	672
Определение местонахождения сборки исполняющей средой .NET Core	674
Резюме	675
<b>Глава 17. Рефлексия типов, позднее связывание и программирование на основе атрибутов</b>	676
Потребность в метаданных типов	676
Просмотр (частичных) метаданных для перечисления EngineStateEnum	677
Просмотр (частичных) метаданных для типа Car	678
Исследование блока TypeRef	680
Документирование определяемой сборки	680
Документирование ссылаемых сборок	680
Документирование строковых литералов	681
Понятие рефлексии	681
Класс System.Type	682
Получение информации о типе с помощью System.Object.GetType()	683
Получение информации о типе с помощью typeof()	684
Получение информации о типе с помощью System.Type.GetType()	684
Построение специального средства для просмотра метаданных	685
Рефлексия методов	685
Рефлексия полей и свойств	686
Рефлексия реализованных интерфейсов	687
Отображение разнообразных дополнительных деталей	687
Добавление операторов верхнего уровня	688
Рефлексия статических типов	689
Рефлексия обобщенных типов	689
Рефлексия параметров и возвращаемых значений методов	690
Динамическая загрузка сборки	691
Рефлексия сборки инфраструктуры	693
Понятие позднего связывания	695
Класс System.Activator	695
Вызов методов без параметров	696
Вызов методов с параметрами	697
Роль атрибутов .NET	698
Потребители атрибутов	699
Применение атрибутов в C#	699
Сокращенная система обозначения атрибутов C#	700
Указание параметров конструктора для атрибутов	701
Атрибут [Obsolete] в действии	701

Построение специальных атрибутов	702
Применение специальных атрибутов	703
Синтаксис именованных свойств	704
Ограничение использования атрибутов	704
Атрибуты уровня сборки	705
Использование файла проекта для атрибутов сборки	706
Рефлексия атрибутов с использованием раннего связывания	707
Рефлексия атрибутов с использованием позднего связывания	708
Практическое использование рефлексии, позднего связывания и специальных атрибутов	709
Построение расширяемого приложения	710
Построение мультипроектного решения ExtendableApp	711
Построение сборки CommonSnappableTypes.dll	714
Построение оснастки на C#	715
Построение оснастки на Visual Basic	715
Добавление кода для ExtendableApp	716
Резюме	718
<b>Глава 18. Динамические типы и среда DLR</b>	<b>719</b>
Роль ключевого слова dynamic языка C#	719
Вызов членов на динамически объявленных данных	721
Область использования ключевого слова dynamic	723
Ограничения ключевого слова dynamic	723
Практическое использование ключевого слова dynamic	724
Роль исполняющей среды динамического языка	725
Роль деревьев выражений	725
Динамический поиск в деревьях выражений во время выполнения	726
Упрощение вызовов с поздним связыванием посредством динамических типов	726
Использование ключевого слова dynamic для передачи аргументов	727
Упрощение взаимодействия с COM посредством динамических данных (только Windows)	729
Роль основных сборок взаимодействия	730
Встраивание метаданных взаимодействия	731
Общие сложности взаимодействия с COM	732
Взаимодействие с COM с использованием динамических данных C#	732
Резюме	736
<b>Глава 19. Язык CIL и роль динамических сборок</b>	<b>737</b>
Причины для изучения грамматики языка CIL	737
Директивы, атрибуты и коды операций CIL	739
Роль директив CIL	739
Роль атрибутов CIL	739
Роль кодов операций CIL	740
Разница между кодами операций и их мнемоническими эквивалентами в CIL	740
Заталкивание и выталкивание: основанная на стеке природа CIL	741
Возвратное проектирование	743
Роль меток в коде CIL	745
Взаимодействие с CIL: модификация файла *.il	746
Компиляция кода CIL	746

Директивы и атрибуты CIL	747
Указание ссылок на внешние сборки в CIL	747
Определение текущей сборки в CIL	748
Определение пространств имен в CIL	749
Определение типов классов в CIL	749
Определение и реализация интерфейсов в CIL	751
Определение структур в CIL	751
Определение перечислений в CIL	752
Определение обобщений в CIL	752
Компиляция файла CILTypes.il	753
Соответствия между типами данных в библиотеке базовых классов .NET Core, C# и CIL	753
Определение членов типов в CIL	754
Определение полей данных в CIL	754
Определение конструкторов типа в CIL	755
Определение свойств в CIL	755
Определение параметров членов	756
Исследование кодов операций CIL	756
Директива .maxstack	759
Объявление локальных переменных в CIL	759
Отображение параметров на локальные переменные в CIL	760
Скрытая ссылка this	760
Представление итерационных конструкций в CIL	761
Заключительные слова о языке CIL	762
Динамические сборки	762
Исследование пространства имен System.Reflection.Emit	763
Роль типа System.Reflection.Emit.ILGenerator	763
Выпуск динамической сборки	764
Выпуск сборки и набора модулей	767
Роль типа ModuleBuilder	768
Выпуск типа HelloClass и строковой переменной-члена	769
Выпуск конструкторов	769
Выпуск метода SayHello()	770
Использование динамически сгенерированной сборки	770
Резюме	771
<b>Часть VI. Работа с файлами, сериализация объектов и доступ к данным</b>	<b>773</b>
<b>Глава 20. Файловый ввод-вывод и сериализация объектов</b>	<b>774</b>
Исследование пространства имен System.IO	774
Классы Directory (DirectoryInfo) и File (FileInfo)	776
Абстрактный базовый класс FileSystemInfo	776
Работа с типом DirectoryInfo	777
Перечисление файлов с помощью типа DirectoryInfo	779
Создание подкаталогов с помощью типа DirectoryInfo	779
Работа с типом Directory	780
Работа с типом DriveInfo	781

Работа с типом FileInfo	782
Метод FileInfo.Create()	783
Метод FileInfo.Open()	784
Методы FileInfo.OpenRead() и FileInfo.OpenWrite()	785
Метод FileInfo.OpenText()	786
Методы FileInfo.CreateText() и FileInfo.AppendText()	786
Работа с типом File	787
Дополнительные члены типа File	788
Абстрактный класс Stream	789
Работа с типом FileStream	790
Работа с типами StreamWriter и StreamReader	791
Запись в текстовый файл	792
Чтение из текстового файла	793
Прямое создание объектов типа StreamWriter/StreamReader	793
Работа с типами StringWriter и StringReader	794
Работа с типами BinaryWriter и BinaryReader	795
Программное слежение за файлами	797
Понятие сериализации объектов	799
Роль графов объектов	800
Создание примеров типов и написание операторов верхнего уровня	801
Сериализация и десериализация с помощью XmlSerializer	803
Сериализация и десериализация с помощью System.Text.Json	807
Резюме	814
<b>Глава 21. Доступ к данным с помощью ADO.NET</b>	815
Сравнение ADO.NET и ADO	816
Поставщики данных ADO.NET	816
Поставщики данных ADO.NET	818
Типы из пространства имен System.Data	818
Роль интерфейса IDbConnection	819
Роль интерфейса IDbTransaction	820
Роль интерфейса IDbCommand	820
Роль интерфейсов IDbDataParameter и IDataParameter	820
Роль интерфейсов IDbDataAdapter и IDataAdapter	821
Роль интерфейсов IDataReader и IDataRecord	822
Абстрагирование поставщиков данных с использованием интерфейсов	823
Установка SQL Server и Azure Data Studio	825
Установка SQL Server	826
Установка IDE-среды SQL Server	828
Подключение к SQL Server	828
Восстановление базы данных AutoLot из резервной копии	830
Копирование файла резервной копии в имеющийся контейнер	830
Восстановление базы данных с помощью SSMS	831
Восстановление базы данных с помощью Azure Data Studio	832
Создание базы данных AutoLot	832
Создание базы данных	833
Создание таблиц	833
Создание отношений между таблицами	835
Создание хранимой процедуры GetPetName	837

Добавление тестовых записей	838
Модель фабрики поставщиков данных ADO.NET	839
Полный пример фабрики поставщиков данных	840
Потенциальный недостаток модели фабрики поставщиков данных	844
Погружение в детали объектов подключений, команд и чтения данных	845
Работа с объектами подключений	846
Работа с объектами команд	849
Работа с объектами чтения данных	850
Работа с запросами создания, обновления и удаления	853
Создание классов Car и CarViewModel	853
Добавление класса InventoryDal	854
Добавление реализации IDisposable	855
Создание строго типизированного метода InsertCar()	858
Добавление логики удаления	858
Добавление логики обновления	859
Работа с параметризованными объектами команд	860
Выполнение хранимой процедуры	863
Создание консольного клиентского приложения	865
Понятие транзакций базы данных	866
Основные члены объекта транзакции ADO.NET	867
Добавление метода транзакции в InventoryDal	867
Тестирование транзакции базы данных	870
Выполнение массового копирования с помощью ADO.NET	871
Исследование класса SqlBulkCopy	871
Создание специального класса чтения данных	871
Выполнение массового копирования	875
Тестирование массового копирования	876
Резюме	877
<b>Часть VII. Entity Framework Core</b>	<b>879</b>
<b>Глава 22. Введение в Entity Framework Core</b>	<b>880</b>
Инструменты объектно-реляционного отображения	881
Роль Entity Framework Core	882
Строительные блоки Entity Framework Core	883
Класс DbContext	883
Поддержка транзакций и точек сохранения	887
Транзакции и стратегии выполнения	887
Класс DbSet<T>	888
Экземпляр ChangeTracker	891
Сущности	891
Выполнение запросов	912
Смешанное выполнение на клиентской и серверной сторонах	913
Сравнение отслеживаемых и неотслеживаемых запросов	913
Важные функциональные средства EF Core	914
Обработка значений, генерируемых базой данных	914
Проверка параллелизма	915
Устойчивость подключений	916
Связанные данные	917

Глобальные фильтры запросов	920
Выполнение низкоуровневых запросов SQL с помощью LINQ	922
Пакетирование операторов	923
Принадлежащие сущностные типы	924
Сопоставление с функциями базы данных	927
Команды CLI глобального инструмента EF Core	927
Команды для управления миграциями	929
Команды для управления базой данных	933
Команды для управления типами DbContext	934
Резюме	936
<b>Глава 23. Построение уровня доступа к данным с помощью Entity Framework Core</b>	<b>937</b>
“Сначала код” или “сначала база данных”	937
Создание проектов <code>AutoLot.Dal</code> и <code>AutoLot.Models</code>	938
Создание шаблонов для класса, производного от <code>DbContext</code> , и сущностных классов	939
Переключение на подход “сначала код”	940
Создание фабрики экземпляров класса, производного от <code>DbContext</code> , на этапе проектирования	940
Создание начальной миграции	941
Применение миграции	941
Обновление модели	942
Сущности	942
Класс <code>ApplicationDbContext</code>	951
Создание миграции и обновление базы данных	957
Добавление представления базы данных и хранимой процедуры	958
Добавление класса <code>MigrationHelpers</code>	958
Обновление и применение миграции	959
Добавление модели представления	960
Добавление класса модели представления	960
Добавление класса модели представления к <code>ApplicationDbContext</code>	961
Добавление хранилищ	961
Добавление базового интерфейса <code>IRepo</code>	962
Добавление класса <code>BaseRepo</code>	962
Интерфейсы хранилищ, специфичных для сущностей	966
Реализация классов хранилищ, специфичных для сущностей	967
Программная работа с базой данных и миграциями	972
Удаление, создание и очистка базы данных	973
Инициализация базы данных	974
Создание выборочных данных	974
Загрузка выборочных данных	975
Настройка тестов	977
Создание проекта	977
Конфигурирование проекта	978
Создание класса <code>TestHelpers</code>	978
Добавление класса <code>BaseTest</code>	980
Добавление класса тестовой оснастки <code>EnsureAutoLotDatabase</code>	981

Добавление классов интеграционных тестов	982
Выполнение тестов	985
Запрашивание базы данных	985
Состояние сущности	986
Запросы LINQ	986
Выполнение запросов SQL с помощью LINQ	1004
Методы агрегирования	1005
Any() и All()	1007
Получение данных из хранимых процедур	1008
Создание записей	1009
Состояние сущности	1009
Добавление одной записи	1009
Добавление одной записи с использованием метода Attach()	1010
Добавление нескольких записей одновременно	1011
Соображения относительно столбца идентичности при добавлении записей	1012
Добавление объектного графа	1012
Обновление записей	1013
Состояние сущности	1013
Обновление отслеживаемых сущностей	1013
Обновление неотслеживаемых сущностей	1014
Проверка параллелизма	1015
Удаление записей	1016
Состояние сущности	1016
Удаление отслеживаемых сущностей	1017
Удаление неотслеживаемых сущностей	1017
Перехват отказов каскадного удаления	1018
Проверка параллелизма	1018
Резюме	1018
<b>Часть VIII. Разработка клиентских приложений для Windows</b>	<b>1019</b>
<b>Глава 24. Введение в Windows Presentation Foundation и XAML</b>	<b>1020</b>
Побудительные причины создания WPF	1020
Унификация несходных API-интерфейсов	1021
Обеспечение разделения обязанностей через XAML	1022
Обеспечение оптимизированной модели визуализации	1022
Упрощение программирования сложных пользовательских интерфейсов	1023
Исследование сборок WPF	1024
Роль класса Application	1026
Построение класса приложения	1026
Перечисление элементов коллекции Windows	1027
Роль класса Window	1027
Синтаксис XAML для WPF	1032
Введение в <code>CaXaml</code>	1032
Пространства имен XML и “ключевые слова” XAML	1034
Управление видимостью классов и переменных-членов	1036
Элементы XAML, атрибуты XAML и преобразователи типов	1037
Понятие синтаксиса “свойство-элемент” в XAML	1038
Понятие присоединяемых свойств XAML	1039

Понятие расширений разметки XAML	1040
Построение приложений WPF с использованием Visual Studio	1042
Шаблоны проектов WPF	1042
Панель инструментов и визуальный конструктор/редактор XAML	1042
Установка свойств с использованием окна Properties	1043
Обработка событий с использованием окна Properties	1044
Обработка событий в редакторе XAML	1045
Окно Document Outline	1046
Включение и отключение отладчика XAML	1046
Исследование файла App.xaml	1047
Отображение разметки XAML окна на код C#	1048
Роль BAML	1050
Разгадывание загадки Main()	1051
Взаимодействие с данными уровня приложения	1051
Обработка закрытия объекта Window	1052
Перехват событий мыши	1053
Перехват событий клавиатуры	1054
Резюме	1055
<b>Глава 25. Элементы управления, компоновки, события и привязка данных в WPF</b>	<b>1056</b>
Обзор основных элементов управления WPF	1056
Элементы управления для работы с Ink API	1057
Элементы управления для работы с документами WPF	1058
Общие диалоговые окна WPF	1058
Краткий обзор визуального конструктора WPF в Visual Studio	1058
Работа с элементами управления WPF в Visual Studio	1059
Работа с окном Document Outline	1060
Управление компоновкой содержимого с использованием панелей	1060
Позиционирование содержимого внутри панелей Canvas	1062
Позиционирование содержимого внутри панелей WrapPanel	1064
Позиционирование содержимого внутри панелей StackPanel	1065
Позиционирование содержимого внутри панелей Grid	1066
Панели Grid с типами GridSplitter	1068
Позиционирование содержимого внутри панелей DockPanel	1069
Включение прокрутки в типах панелей	1070
Конфигурирование панелей с использованием визуальных конструкторов Visual Studio	1071
Построение окна с использованием вложенных панелей	1073
Построение системы меню	1074
Визуальное построение меню	1076
Построение панели инструментов	1076
Построение строки состояния	1077
Завершение проектирования пользовательского интерфейса	1077
Реализация обработчиков событий MouseEnter/MouseLeave	1078
Реализация логики проверки правописания	1078
Понятие команд WPF	1079
Внутренние объекты команд	1080

Подключение команд к свойству Command	1081
Подключение команд к произвольным действиям	1081
Работа с командами Open и Save	1082
Понятие маршрутизируемых событий	1084
Роль пузырьковых маршрутизируемых событий	1086
Продолжение или прекращение пузырькового распространения	1087
Роль туннельных маршрутизируемых событий	1087
Более глубокое исследование API-интерфейсов и элементов управления WPF	1089
Работа с элементом управления TabControl	1089
Построение вкладки Ink API	1090
Проектирование панели инструментов	1090
Элемент управления RadioButton	1091
Добавление кнопок сохранения, загрузки и удаления	1091
Добавление элемента управления InkCanvas	1092
Предварительный просмотр окна	1092
Обработка событий для вкладки Ink API	1092
Добавление элементов управления в панель инструментов	1093
Элемент управления InkCanvas	1094
Элемент управления ComboBox	1095
Сохранение, загрузка и очистка данных InkCanvas	1097
Введение в модель привязки данных WPF	1098
Построение вкладки Data Binding	1099
Установка привязки данных	1099
Свойство DataContext	1100
Форматирование привязанных данных	1101
Преобразование данных с использованием интерфейса IValueConverter	1101
Установление привязок данных в коде	1103
Построение вкладки DataGrid	1104
Роль свойств зависимости	1106
Исследование существующего свойства зависимости	1108
Важные замечания относительно оболочек свойств CLR	1111
Построение специального свойства зависимости	1111
Добавление процедуры проверки достоверности данных	1114
Реагирование на изменение свойства	1115
Резюме	1116
<b>Глава 26. Службы визуализации графики WPF</b>	1117
Понятие служб визуализации графики WPF	1117
Варианты графической визуализации WPF	1118
Визуализация графических данных с использованием фигур	1119
Добавление прямоугольников, эллипсов и линий на поверхность Canvas	1120
Удаление прямоугольников, эллипсов и линий с поверхности Canvas	1124
Работа с элементами Polyline и Polygon	1125
Работа с элементом Path	1125
Кисти и перья WPF	1129
Конфигурирование кистей с использованием Visual Studio	1129
Конфигурирование кистей в коде	1131
Конфигурирование перьев	1132

Применение графических трансформаций	1132
Первый взгляд на трансформации	1133
Трансформация данных Canvas	1134
Работа с редактором трансформаций Visual Studio	1136
Построение начальной компоновки	1136
Применение трансформаций на этапе проектирования	1138
Трансформация холста в коде	1139
Визуализация графических данных с использованием рисунков и геометрических объектов	1140
Построение кисти DrawingBrush с использованием геометрических объектов	1141
Рисование с помощью DrawingBrush	1142
Включение типов Drawing в DrawingImage	1142
Работа с векторными изображениями	1143
Преобразование файла с векторной графикой в файл XAML	1143
Импортирование графических данных в проект WPF	1144
Взаимодействие с изображением	1145
Визуализация графических данных с использованием визуального уровня	1145
Базовый класс Visual и производные дочерние классы	1146
Первый взгляд на класс DrawingVisual	1146
Визуализация графических данных в специальном диспетчере компоновки	1148
Реагирование на операции проверки попадания	1151
Резюме	1152
<b>Глава 27. Ресурсы, анимация, стили и шаблоны WPF</b>	<b>1153</b>
Система ресурсов WPF	1153
Работа с двоичными ресурсами	1154
Работа с объектными (логическими) ресурсами	1158
Роль свойства Resources	1158
Определение ресурсов уровня окна	1159
Расширение разметки {StaticResource}	1161
Расширение разметки {DynamicResource}	1162
Ресурсы уровня приложения	1162
Определение объединенных словарей ресурсов	1163
Определение сборки, включающей только ресурсы	1164
Службы анимации WPF	1165
Роль классов анимации	1166
Свойства To, From и By	1166
Роль базового класса Timeline	1167
Реализация анимации в коде C#	1167
Управление темпом анимации	1169
Запуск в обратном порядке и циклическое выполнение анимации	1170
Реализация анимации в разметке XAML	1170
Роль раскадровок	1171
Роль триггеров событий	1172
Анимация с использованием дискретных ключевых кадров	1172
Роль стилей WPF	1173
Определение и применение стиля	1174
Переопределение настроек стиля	1175

Влияние атрибута TargetType на стили	1175
Создание подклассов существующих стилей	1176
Определение стилей с триггерами	1177
Определение стилей с множеством триггеров	1178
Стили с анимацией	1178
Применение стилей в коде	1179
Логические деревья, визуальные деревья и стандартные шаблоны	1180
Программное инспектирование логического дерева	1181
Программное инспектирование визуального дерева	1183
Программное инспектирование стандартного шаблона элемента управления	1183
Построение шаблона элемента управления с помощью инфраструктуры триггеров	1186
Шаблоны как ресурсы	1187
Встраивание визуальных подсказок с использованием триггеров	1188
Роль расширения разметки {TemplateBinding}	1189
Роль класса ContentPresenter	1190
Встраивание шаблонов в стили	1190
Резюме	1191
<b>Глава 28. Уведомления WPF, проверка достоверности, команды и MVVM</b>	<b>1193</b>
Введение в паттерн MVVM	1193
Модель	1194
Представление	1194
Модель представления	1194
Анемичные модели или анемичные модели представлений	1194
Система уведомлений привязки WPF	1195
Наблюдаемые модели и коллекции	1195
Добавление привязок и данных	1197
Изменение данных об автомобиле в коде	1198
Наблюдаемые модели	1199
Наблюдаемые коллекции	1201
Итоговые сведения об уведомлениях и наблюдаемых моделях	1203
Проверка достоверности WPF	1204
Модификация примера для демонстрации проверки достоверности	1204
Класс Validation	1204
Варианты проверки достоверности	1205
Использование аннотаций данных в WPF	1215
Настройка свойства ErrorTemplate	1217
Итоговые сведения о проверке достоверности	1219
Создание специальных команд	1219
Реализация интерфейса ICommand	1219
Добавление класса ChangeColorCommand	1220
Создание класса CommandBase	1222
Добавление класса AddCarCommand	1223
Объекты RelayCommand	1224
Итоговые сведения о командах	1226
Перенос кода и данных в модель представления	1226
Перенос кода MainWindow.xaml.cs	1227

Обновление кода и разметки MainWindow	1227
Обновление разметки элементов управления	1228
Итоговые сведения о моделях представлений	1228
Обновление проекта AutoLot .Dal для MVVM	1229
Резюме	1229
<b>Часть IX. ASP.NET Core</b>	1231
<b>Глава 29. Введение в ASP.NET Core</b>	1232
Краткий экскурс в прошлое	1232
Введение в паттерн MVC	1232
ASP.NET Core и паттерн MVC	1233
ASP.NET Core и .NET Core	1233
Одна инфраструктура, много сценариев использования	1234
Функциональные средства ASP.NET Core из MVC/Web API	1234
Соглашения по конфигурации	1235
Привязка моделей	1238
Проверка достоверности моделей	1242
Маршрутизация	1243
Фильтры	1249
Нововведения в ASP.NET Core	1250
Встроенное внедрение зависимостей	1250
Осведомленность о среде	1251
Конфигурация приложений	1253
Развертывание приложений ASP.NET Core	1254
Легковесный и модульный конвейер запросов HTTP	1254
Создание и конфигурирование решения	1254
Использование Visual Studio	1254
Использование командной строки	1257
Запуск приложений ASP.NET Core	1258
Конфигурирование настроек запуска	1259
Использование Visual Studio	1259
Использование командной строки или окна терминала Visual Studio Code	1260
Использование Visual Studio Code	1260
Отладка приложений ASP.NET Core	1261
Обновление портов AutoLot .Api	1262
Создание и конфигурирование экземпляра WebHost	1262
Файл Program.cs	1262
Файл Startup.cs	1263
Ведение журнала	1270
Резюме	1283
<b>Глава 30. Создание служб REST с помощью ASP.NET Core</b>	1284
Введение в REST-службы ASP.NET Core	1284
Создание действий контроллера с использованием служб REST	1285
Результаты ответов в формате JSON	1285
Атрибут ApiController	1287
Обновление настроек Swagger/OpenAPI	1290
Обновление обращений к Swagger в классе Startup	1290

Добавление файла XML-документации	1291
Добавление XML-комментариев в процесс генерации Swagger	1293
Дополнительные возможности документирования для конечных точек API	1294
Построение методов действий API	1296
Конструктор	1297
Методы GetXXX( )	1297
Метод UpdateOne( )	1298
Метод AddOne( )	1299
Метод DeleteOne( )	1300
Класс CarsController	1301
Оставшиеся контроллеры	1302
Фильтры исключений	1304
Создание специального фильтра исключений	1305
Тестирование фильтра исключений	1307
Добавление поддержки запросов между источниками	1307
Создание политики CORS	1307
Добавление политики CORS в конвейер обработки HTTP	1308
Резюме	1308
<b>Глава 31. Создание приложений MVC с помощью ASP.NET Core</b>	<b>1309</b>
Введение в представления ASP.NET Core	1309
Экземпляры класса ViewResult и методы действий	1309
Механизм визуализации и синтаксис Razor	1312
Представления	1315
Компоновки	1319
Частичные представления	1320
Обновление компоновки с использованием частичных представлений	1321
Отправка данных представлениям	1322
Вспомогательные функции дескрипторов	1324
Включение вспомогательных функций дескрипторов	1324
Вспомогательная функция дескриптора для формы	1330
Вспомогательная функция дескриптора для действия формы	1331
Вспомогательная функция дескриптора для якоря	1332
Вспомогательная функция дескриптора для элемента ввода	1332
Вспомогательная функция дескриптора для текстовой области	1333
Вспомогательная функция дескриптора для элемента выбора	1333
Вспомогательные функции дескрипторов для проверки достоверности	1334
Вспомогательная функция дескриптора для среды	1335
Вспомогательная функция дескриптора для ссылки	1336
Вспомогательная функция дескриптора для сценария	1336
Вспомогательная функция дескриптора для изображения	1338
Специальные вспомогательные функции дескрипторов	1338
Подготовительные шаги	1338
Создание базового класса	1339
Вспомогательная функция дескриптора для вывода сведений об элементе	1340
Вспомогательная функция дескриптора для удаления элемента	1341
Вспомогательная функция дескриптора для редактирования сведений об элементе	1342
Вспомогательная функция дескриптора для создания элемента	1342

Вспомогательная функция дескриптора для вывода списка элементов	1343
Обеспечение видимости специальных вспомогательных функций дескрипторов	1344
Вспомогательные функции HTML	1344
Вспомогательная функция <code>DisplayFor()</code>	1345
Вспомогательная функция <code>DisplayForModel()</code>	1345
Вспомогательные функции <code>EditorFor()</code> и <code>EditorForModel()</code>	1345
Управление библиотеками клиентской стороны	1345
Установка диспетчера библиотек как глобального инструмента <code>.NET Core</code>	1346
Добавление в проект <code>AutoLot.Mvc</code> библиотек клиентской стороны	1346
Завершение работы над представлениями <code>CarsController</code> и <code>Cars</code>	1350
Класс <code>CarsController</code>	1350
Частичное представление списка автомобилей	1351
Представление <code>Index</code>	1352
Представление <code>ByMake</code>	1353
Представление <code>Details</code>	1354
Представление <code>Create</code>	1356
Представление <code>Edit</code>	1358
Представление <code>Delete</code>	1360
Компоненты представлений	1361
Код серверной стороны	1362
Построение частичного представления	1363
Вызов компонентов представлений	1364
Вызов компонентов представлений как специальных вспомогательных функций дескрипторов	1364
Обновление меню	1364
Пакетирование и минификация	1365
Пакетирование	1365
Минификация	1365
Решение <code>WebOptimizer</code>	1365
Шаблон параметров в <code>ASP.NET Core</code>	1367
Добавление информации об автодилере	1367
Создание оболочки службы	1369
Обновление конфигурации приложения	1369
Создание класса <code>ApiServiceSettings</code>	1370
Оболочка службы API	1370
Конфигурирование служб	1374
Построение класса <code>CarsController</code>	1375
Вспомогательный метод <code>GetMakes()</code>	1376
Вспомогательный метод <code>GetOneCar()</code>	1376
Открытые методы действий	1376
Обновление компонента представления	1378
Совместный запуск приложений <code>AutoLot.Mvc</code> и <code>AutoLot.Api</code>	1379
Использование <code>Visual Studio</code>	1379
Использование командной строки	1380
Резюме	1380
<b>Предметный указатель</b>	<b>1381</b>

## ГЛАВА 14

# Процессы, домены приложений и контексты загрузки

В настоящей главе будут представлены детали обслуживания сборки исполняющей средой, а также отношения между процессами, доменами приложений и контекстами загрузки.

Выражаясь кратко, *домены приложений* (Application Domain или просто AppDomain) представляют собой логические подразделы внутри заданного процесса, обслуживающего набор связанных сборок .NET Core. Как вы увидите, каждый домен приложения в дальнейшем подразделяется на *контекстные границы*, которые используются для группирования вместе похожих по смыслу объектов .NET Core. Благодаря понятию контекста исполняющая среда способна обеспечивать надлежащую обработку объектов со специальными требованиями.

Хотя вполне справедливо утверждать, что многие повседневные задачи программирования не предусматривают работу с процессами, доменами приложений или контекстами загрузки напрямую, их понимание важно при взаимодействии с многочисленными API-интерфейсами .NET Core, включая многопоточную и параллельную обработку, а также сериализацию объектов.

## Роль процесса Windows

Концепция “процесса” существовала в операционных системах Windows задолго до выпуска платформы .NET/.NET Core. Пользуясь простыми терминами, *процесс* — это выполняющаяся программа. Тем не менее, формально процесс является концепцией уровня операционной системы, которая применяется для описания набора ресурсов (таких как внешние библиотеки кода и главный поток) и необходимых распределений памяти, используемой функционирующим приложением. Для каждого загруженного в память приложения .NET Core операционная система создает отдельный изолированный процесс для применения на протяжении всего времени его существования.

При использовании такого подхода к изоляции приложений в результате получается намного более надежная и устойчивая исполняющая среда, поскольку отказ одного процесса не влияет на работу других процессов. Более того, данные в одном процессе не доступны напрямую другим процессам, если только не применяются специфичные инструменты вроде пространства имен System.IO.Pipes или класса MemoryMappedFile.

Каждый процесс Windows получает уникальный идентификатор процесса (process identifier — PID) и может по мере необходимости независимо загружаться и выгружаться операционной системой (а также программно). Как вам возможно известно, в окне диспетчера задач Windows (открываемом по нажатию комбинации клавиш <Ctrl+Shift+Esc>) имеется вкладка Processes (Процессы), на которой можно просматривать разнообразные статические данные о процессах, функционирующих на машине. На вкладке Details (Подробности) можно видеть назначенный идентификатор PID и имя образа (рис. 14.1).

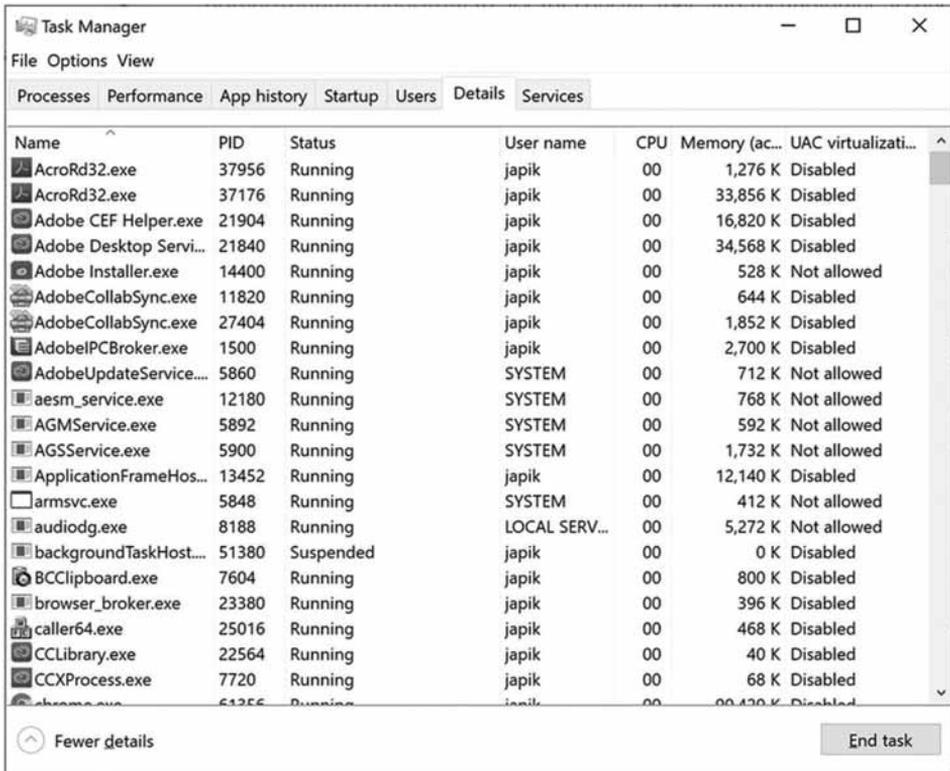


Рис. 14.1. Диспетчер задач Windows

## Роль потоков

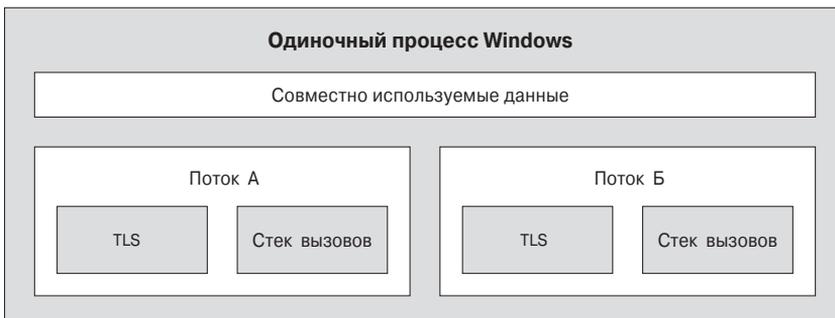
Каждый процесс Windows содержит начальный “поток”, который действует как точка входа для приложения. Особенности построения многопоточных приложений на платформе .NET Core рассматриваются в главе 15; однако для понимания материала настоящей главы необходимо ознакомиться с несколькими рабочими определениями. Поток представляет собой путь выполнения внутри процесса. Выражаясь формально, первый поток, созданный точкой входа процесса, называется *главным потоком*. В любой программе .NET Core (консольном приложении, Windows-службе, приложении WPF и т.д.) точка входа помечается с помощью метода `Main()` или файла, содержащего операторы верхнего уровня. При обращении к этому коду автоматически создается главный поток.

Процессы, которые содержат единственный главный поток выполнения, по своей сути *безопасны в отношении потоков*, т.к. в каждый момент времени доступ к данным приложения может получать только один поток. Тем не менее, однопоточный процесс (особенно с графическим пользовательским интерфейсом) часто замедленно реагирует на действия пользователя, когда его единственный поток выполняет сложную операцию (наподобие печати длинного текстового файла, сложных математических вычислений или попытки подключения к удаленному серверу, находящемуся на расстоянии тысяч километров).

Учитывая такой потенциальный недостаток однопоточных приложений, операционные системы, которые поддерживаются .NET Core, и сама платформа .NET Core предоставляют главному потоку возможность порождения дополнительных вторичных потоков (называемых *рабочими потоками*) с использованием нескольких функций из API-интерфейса Windows, таких как `CreateThread()`. Каждый поток (первичный или вторичный) становится уникальным путем выполнения в процессе и имеет параллельный доступ ко всем совместно используемым элементам данных внутри этого процесса.

Нетрудно догадаться, что разработчики обычно создают дополнительные потоки для улучшения общей степени отзывчивости программы. Многопоточные процессы обеспечивают иллюзию того, что выполнение многочисленных действий происходит более или менее одновременно. Например, приложение может порождать дополнительный рабочий поток для выполнения трудоемкой единицы работы (вроде вывода на печать крупного текстового файла). После запуска вторичного потока главный поток продолжает реагировать на пользовательский ввод, что дает всему процессу возможность достигать более высокой производительности. Однако на самом деле так происходит не всегда: применение слишком большого количества потоков в одном процессе может приводить к *ухудшению* производительности из-за того, что центральный процессор должен переключаться между активными потоками внутри процесса (а это отнимает время).

На некоторых машинах многопоточность по большей части является иллюзией, обеспечиваемой операционной системой. Машины с единственным (не поддерживающим гиперпотоки) центральным процессором не обладают возможностью обработки множества потоков в одно и то же время. Взамен один центральный процессор выполняет по одному потоку за единицу времени (называемую *квантом времени*), частично основываясь на приоритете потока. По истечении выделенного кванта времени выполнение существующего потока приостанавливается, позволяя выполнять работу другому потоку. Чтобы поток не “забывал”, что происходило до того, как его выполнение было приостановлено, ему предоставляется возможность записывать данные в локальное хранилище потоков (Thread Local Storage — TLS) и выделяется отдельный стек вызовов (рис. 14.2).



**Рис. 14.2.** Отношения между потоками и процессами в Windows

Если тема потоков для вас нова, то не стоит беспокоиться о деталях. На данном этапе просто запомните, что любой поток представляет собой уникальный путь выполнения внутри процесса Windows. Каждый процесс имеет главный поток (созданный посредством точки входа исполняемого файла) и может содержать дополнительные потоки, которые создаются программно.

## Взаимодействие с процессами, используя платформу .NET Core

Несмотря на то что с процессами и потоками не связано ничего нового, способ взаимодействия с ними в рамках платформы .NET Core значительно изменился (в лучшую сторону). Чтобы подготовить почву для понимания области построения многопоточных сборок (см. главу 15), давайте начнем с выяснения способов взаимодействия с процессами, используя библиотеки базовых классов .NET Core.

В пространстве имен `System.Diagnostics` определено несколько типов, которые позволяют программно взаимодействовать с процессами и разнообразными типами, связанными с диагностикой, такими как журнал событий системы и счетчики производительности. В текущей главе нас интересуют только типы, связанные с процессами, которые описаны в табл. 14.1.

**Таблица 14.1. Избранные типы пространства имен `System.Diagnostics`**

Тип, связанный с процессами	Описание
<code>Process</code>	Предоставляет доступ к локальным и удаленным процессам, а также позволяет программно запускать и останавливать процессы
<code>ProcessModule</code>	Представляет модуль (*.dll или *.exe), загруженный в определенный процесс. Важно понимать, что тип <code>ProcessModule</code> может представлять <i>любой</i> модуль, т.е. двоичные сборки, основанные на COM, .NET или традиционном языке C
<code>ProcessModuleCollection</code>	Предоставляет строго типизированную коллекцию объектов <code>ProcessModule</code>
<code>ProcessStartInfo</code>	Указывает набор значений, применяемых при запуске процесса с помощью метода <code>Process.Start()</code>
<code>ProcessThread</code>	Представляет поток внутри заданного процесса. Имейте в виду, что тип <code>ProcessThread</code> используется для диагностирования набора потоков процесса, но не для порождения новых потоков выполнения в рамках процесса
<code>ProcessThreadCollection</code>	Предоставляет строго типизованную коллекцию объектов <code>ProcessThread</code>

Класс `System.Diagnostics.Process` позволяет анализировать процессы, выполняющиеся на заданной машине (локальные или удаленные). В классе `Process` также определены члены, предназначенные для программного запуска и завершения процессов, просмотра (или модификации) уровня приоритета процесса и получения списка активных потоков и/или загруженных модулей внутри указанного процесса. В табл. 14.2 перечислены некоторые основные свойства класса `System.Diagnostics.Process`.

**Таблица 14.2. Избранные свойства класса Process**

Свойство	Описание
ExitTime	Позволяет извлекать отметку времени, ассоциированную с процессом, который был завершен (представленную с помощью типа <code>DateTime</code> )
Handle	Возвращает дескриптор (представленный типом <code>IntPtr</code> ), который был назначен процессу операционной системой. Это может быть полезно при построении приложений .NET, нуждающихся во взаимодействии с неуправляемым кодом
Id	Позволяет получать идентификатор PID связанного процесса
MachineName	Позволяет получать имя компьютера, на котором выполняется связанный процесс
MainWindowTitle	Позволяет получать заголовок главного окна процесса (если у процесса нет главного окна, то возвращается пустая строка)
Modules	Предоставляет доступ к строго типизированной коллекции <code>ProcessModuleCollection</code> , представляющей набор модулей (*.dll или *.exe), которые были загружены внутри текущего процесса
ProcessName	Позволяет получать имя процесса (которое, как и можно было предполагать, представляет собой имя самого приложения)
Responding	Позволяет получать значение, которое указывает, реагирует ли пользовательский интерфейс процесса на пользовательский ввод (или в текущий момент находится в “зависшем” состоянии)
StartTime	Позволяет получать значение времени, когда был запущен процесс (представленное с помощью типа <code>DateTime</code> )
Threads	Позволяет получать набор потоков, выполняемых в связанном процессе (представленный посредством коллекции объектов <code>ProcessThread</code> )

Кроме перечисленных выше свойств в классе `System.Diagnostics.Process` определено несколько полезных методов (табл. 14.3).

**Таблица 14.3. Избранные методы класса Process**

Метод	Описание
<code>CloseMainWindow()</code>	Этот метод закрывает процесс, который содержит пользовательский интерфейс, отправляя его главному окну сообщение о закрытии
<code>GetCurrentProcess()</code>	Этот статический метод возвращает новый объект <code>Process</code> , который представляет процесс, активный в текущий момент
<code>GetProcesses()</code>	Этот статический метод возвращает массив объектов <code>Process</code> , представляющих процессы, которые выполняются на заданной машине
<code>Kill()</code>	Этот метод немедленно останавливает связанный процесс
<code>Start()</code>	Этот метод запускает процесс

## Перечисление выполняющихся процессов

Для иллюстрации способа манипулирования объектами `Process` создайте новый проект консольного приложения C# по имени `ProcessManipulator` и определите в классе `Program` следующий вспомогательный статический метод (не забудьте импортировать в файл кода пространства имен `System.Diagnostics` и `System.Linq`):

```
static void ListAllRunningProcesses ()
{
    // Получить все процессы на локальной машине, упорядоченные по PID.
    var runningProcs = from proc in Process.GetProcesses(".")
                       orderby proc.Id select proc;

    // Вывести для каждого процесса идентификатор PID и имя.
    foreach (var p in runningProcs)
    {
        string info = $"-> PID: {p.Id}\tName: {p.ProcessName}";
        Console.WriteLine(info);
    }
    Console.WriteLine("*****\n");
}
```

Статический метод `Process.GetProcesses()` возвращает массив объектов `Process`, которые представляют выполняющиеся процессы на целевой машине (передаваемая методу строка `.` обозначает локальный компьютер). После получения массива объектов `Process` можно обращаться к любым членам, описанным в табл. 14.2 и 14.3. Здесь просто для каждого процесса выводятся идентификатор PID и имя с упорядочением по PID. Модифицируйте операторы верхнего уровня, как показано ниже:

```
using System;
using System.Diagnostics;
using System.Linq;

Console.WriteLine("***** Fun with Processes *****\n");
ListAllRunningProcesses ();
Console.ReadLine ();
```

Запустив приложение, вы увидите список имен и идентификаторов PID для всех процессов на локальной машине. Ниже показана часть вывода (ваш вывод наверняка будет отличаться):

```
***** Fun with Processes *****
-> PID: 0 Name: Idle
-> PID: 4 Name: System
-> PID: 104 Name: Secure System
-> PID: 176 Name: Registry
-> PID: 908 Name: svchost
-> PID: 920 Name: smss
-> PID: 1016 Name: csrss
-> PID: 1020 Name: NVDisplay.Container
-> PID: 1104 Name: wininit
-> PID: 1112 Name: csrss
*****
```

## Исследование конкретного процесса

В дополнение к полному списку всех выполняющихся процессов на заданной машине статический метод `Process.GetProcessById()` позволяет получать одиночный объект `Process` по ассоциированному с ним идентификатору PID. В случае запроса несуществующего PID генерируется исключение `ArgumentException`. Например, чтобы получить объект `Process`, который представляет процесс с PID, равным 30592, можно написать следующий код:

```
// Если процесс с PID, равным 30592, не существует,
// то сгенерируется исключение во время выполнения.
static void GetSpecificProcess()
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(30592);
    }
    catch(ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

К настоящему моменту вы уже знаете, как получить список всех процессов, а также специфический процесс на машине посредством поиска по PID. Наряду с выяснением идентификаторов PID и имен процессов класс `Process` позволяет просматривать набор текущих потоков и библиотек, применяемых внутри заданного процесса. Давайте посмотрим, как это делается.

## Исследование набора потоков процесса

Набор потоков представлен в виде строго типизированной коллекции `ProcessThreadCollection`, которая содержит определенное количество отдельных объектов `ProcessThread`. В целях иллюстрации добавьте к текущему приложению приведенный далее вспомогательный статический метод:

```
static void EnumThreadsForPid(int pID)
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(pID);
    }
    catch(ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
        return;
    }

    // Вывести статистические сведения по каждому потоку в указанном процессе
    Console.WriteLine("Here are the threads used by: {0}",
        theProc.ProcessName);
    ProcessThreadCollection theThreads = theProc.Threads;
```

```

foreach (ProcessThread pt in theThreads)
{
    string info =
        $"{-> Thread ID: {pt.Id}\tStart Time: {pt.StartTime.ToShortTimeString()}\
tPriority:{pt.PriorityLevel}";
    Console.WriteLine(info);
}
Console.WriteLine("*****\n");
}

```

Как видите, свойство `Threads` в типе `System.Diagnostics.Process` обеспечивает доступ к классу `ProcessThreadCollection`. Здесь для каждого потока внутри указанного клиентом процесса выводится назначенный идентификатор потока, время запуска и уровень приоритета. Обновите операторы верхнего уровня в своей программе, чтобы запрашивать у пользователя идентификатор PID процесса, подлежащего исследованию:

```

static void Main(string[] args)
{
    ...
    // Запросить у пользователя PID и вывести набор активных потоков.
    Console.WriteLine("***** Enter PID of process to investigate *****");
    Console.Write("PID: ");
    string pID = Console.ReadLine();
    int theProcID = int.Parse(pID);
    EnumThreadsForPid(theProcID);
    Console.ReadLine();
}

```

После запуска приложения можно вводить PID любого процесса на машине и просматривать имеющиеся внутри него потоки. В следующем выводе показан неполный список потоков, используемых процессом с PID 3804, который (так случилось) обслуживает браузер Edge:

```

***** Enter PID of process to investigate *****
PID: 3804
Here are the threads used by: msedge
-> Thread ID: 3464 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 19420 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 17780 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 22380 Start Time: 01:20 PM Priority: Normal
-> Thread ID: 27580 Start Time: 01:20 PM Priority: -4
...
*****

```

Помимо `Id`, `StartTime` и `PriorityLevel` тип `ProcessThread` содержит дополнительные члены, наиболее интересные из которых перечислены в табл. 14.4.

Прежде чем двигаться дальше, необходимо уяснить, что тип `ProcessThread` не является сущностью, применяемой для создания, приостановки или уничтожения потоков на платформе .NET Core. Тип `ProcessThread` скорее представляет собой средство, позволяющее получать диагностическую информацию по активным потокам Windows внутри выполняющегося процесса. Более подробные сведения о том, как создавать многопоточные приложения с использованием пространства имен `System.Threading`, приводятся в главе 15.

Таблица 14.4. Избранные члены типа `ProcessThread`

Член	Описание
<code>CurrentPriority</code>	Получает текущий приоритет потока
<code>Id</code>	Получает уникальный идентификатор потока
<code>IdealProcessor</code>	Устанавливает предпочитаемый процессор для выполнения заданного потока
<code>PriorityLevel</code>	Получает или устанавливает уровень приоритета потока
<code>ProcessorAffinity</code>	Устанавливает процессоры, на которых может выполняться связанный поток
<code>StartAddress</code>	Получает адрес в памяти функции, вызванной операционной системой, которая запустила данный поток
<code>StartTime</code>	Получает время, когда операционная система запустила поток
<code>ThreadState</code>	Получает текущее состояние потока
<code>TotalProcessorTime</code>	Получает общее время использования процессора данным потоком
<code>WaitReason</code>	Получает причину, по которой поток находится в состоянии ожидания

## Исследование набора модулей процесса

Теперь давайте посмотрим, как реализовать проход по загруженным модулям, которые размещены внутри конкретного процесса. Когда речь идет о процессах, *модуль* — это общий термин, применяемый для описания заданной сборки `*.dll` (или самого файла `*.exe`), которая обслуживается специфичным процессом. Когда производится доступ к коллекции `ProcessModuleCollection` через свойство `Process.Modules`, появляется возможность перечисления *всех модулей*, размещенных внутри процесса: библиотек на основе .NET Core, COM и традиционного языка C. Взгляните на показанный ниже дополнительный вспомогательный метод, который будет перечислять модули в процессе с указанным идентификатором PID:

```
static void EnumModsForPid(int pID)
{
    Process theProc = null;
    try
    {
        theProc = Process.GetProcessById(pID);
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine(ex.Message);
        return;
    }

    Console.WriteLine("Here are the loaded modules for: {0}",
        theProc.ProcessName);
    ProcessModuleCollection theMods = theProc.Modules;
```

```

foreach (ProcessModule pm in theMods)
{
    string info = $"-> Mod Name: {pm.ModuleName}";
    Console.WriteLine(info);
}
Console.WriteLine("*****\n");
}

```

Чтобы получить какой-то вывод, давайте посмотрим загружаемые модули для процесса, обслуживающего программу текущего примера (`ProcessManipulator`). Для этого нужно запустить приложение, выяснить идентификатор PID, назначенный `ProcessManipulator.exe` (посредством диспетчера задач), и передать значение PID методу `EnumModsForPid()`. Вас может удивить, что с простым консольным приложением связан настолько внушительный список библиотек `*.dll` (`GDI32.dll`, `USER32.dll`, `ole32.dll` и т.д.). Ниже показан частичный список загруженных модулей (ради краткости отредактированный):

```

Here are (some of) the loaded modules for: ProcessManipulator
Here are the loaded modules for: ProcessManipulator
-> Mod Name: ProcessManipulator.exe
-> Mod Name: ntdll.dll
-> Mod Name: KERNEL32.DLL
-> Mod Name: KERNELBASE.dll
-> Mod Name: USER32.dll
-> Mod Name: win32u.dll
-> Mod Name: GDI32.dll
-> Mod Name: gdi32full.dll
-> Mod Name: msvc_p_win.dll
-> Mod Name: ucrtbase.dll
-> Mod Name: SHELL32.dll
-> Mod Name: ADVAPI32.dll
-> Mod Name: msvcrt.dll
-> Mod Name: sechost.dll
-> Mod Name: RPCRT4.dll
-> Mod Name: IMM32.DLL
-> Mod Name: hostfxr.dll
-> Mod Name: hostpolicy.dll
-> Mod Name: coreclr.dll
-> Mod Name: ole32.dll
-> Mod Name: combase.dll
-> Mod Name: OLEAUT32.dll
-> Mod Name: bcryptPrimitives.dll
-> Mod Name: System.Private.CoreLib.dll
...
*****

```

## Запуск и останов процессов программным образом

Финальными аспектами класса `System.Diagnostics.Process`, которые мы здесь исследуем, являются методы `Start()` и `Kill()`. Они позволяют программно запускать и завершать процесс. В качестве примера создадим вспомогательный статический метод `StartAndKillProcess()` с приведенным ниже кодом.

**На заметку!** В зависимости от настроек операционной системы, касающихся безопасности, для запуска новых процессов могут требоваться права администратора.

```
static void StartAndKillProcess()
{
    Process proc = null;
    // Запустить Edge и перейти на Facebook!
    try
    {
        proc = Process.Start(@"C:\Program Files (x86)\Microsoft\Edge\
Application\msedge.exe",
            "www.facebook.com");
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine(ex.Message);
    }
    // Уничтожить процесс по нажатию <Enter>.
    Console.WriteLine("--> Hit enter to kill {0}...",
        proc.ProcessName);
    Console.ReadLine();
    // Уничтожить все процессы msedge.exe.
    try
    {
        foreach (var p in Process.GetProcessesByName("MsEdge"))
        {
            p.Kill(true);
        }
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

Статический метод `Process.Start()` имеет несколько перегруженных версий. Как минимум, понадобится указать путь и имя файла запускаемого процесса. В рассматриваемом примере используется версия метода `Start()`, которая позволяет задавать любые дополнительные аргументы, подлежащие передаче в точку входа программы, в данном случае веб-страницу для загрузки.

В результате вызова метода `Start()` возвращается ссылка на новый активизированный процесс. Чтобы завершить данный процесс, потребуется просто вызвать метод `Kill()` уровня экземпляра. Поскольку `Microsoft Edge` запускает множество процессов, для их уничтожения организован цикл. Вызовы `Start()` и `Kill()` помещены внутрь блока `try/catch` с целью обработки исключений `InvalidOperationException`. Это особенно важно при вызове метода `Kill()`, потому что такое исключение генерируется, если процесс был завершён до вызова `Kill()`.