

Содержание

| | |
|--|-----------|
| Об авторах | 27 |
| Благодарности | 28 |
| Предисловие | 29 |
| Предполагаемая читательская аудитория | 29 |
| Как организована эта книга | 30 |
| Что требуется для работы с этой книгой | 30 |
| Соглашения, используемые в этой книге | 30 |
| Использование примеров кода | 31 |
| От издательства | 32 |
| Глава 1. Введение в C# и .NET Framework | 33 |
| Объектная ориентация | 33 |
| Безопасность в отношении типов | 34 |
| Управление памятью | 35 |
| Поддержка платформ | 35 |
| Отношения между C# и CLR | 35 |
| CLR и .NET Framework | 35 |
| Язык C# и Windows Runtime | 37 |
| Нововведения версии C# 6.0 | 38 |
| Нововведения версии C# 5.0 | 40 |
| Нововведения версии C# 4.0 | 40 |
| Нововведения версии C# 3.0 | 41 |
| Глава 2. Основы языка C# | 43 |
| Первая программа на C# | 43 |
| Компиляция | 45 |
| Синтаксис | 46 |
| Идентификаторы и ключевые слова | 46 |
| Литералы, знаки пунктуации и операции | 48 |
| Комментарии | 48 |
| Основы типов | 48 |
| Примеры predefined типов | 49 |
| Примеры специальных типов | 49 |
| Преобразования | 52 |
| Типы значений и ссылочные типы | 52 |
| Классификация predefined типов | 56 |
| Числовые типы | 56 |
| Числовые литералы | 57 |
| Числовые преобразования | 58 |
| Арифметические операции | 59 |
| Операции инкремента и декремента | 59 |
| Специальные целочисленные операции | 60 |
| 8- и 16-битные целочисленные типы | 61 |
| Специальные значения float и double | 62 |
| Выбор между double и decimal | 63 |
| Ошибки округления вещественных чисел | 63 |

| | |
|---|------------|
| Булевские типы и операции | 64 |
| Булевские преобразования | 64 |
| Операции сравнения и проверки равенства | 64 |
| Условные операции | 65 |
| Строки и символы | 65 |
| Символьные преобразования | 66 |
| Строковый тип | 66 |
| Массивы | 68 |
| Стандартная инициализация элементов | 68 |
| Многомерные массивы | 69 |
| Упрощенные выражения инициализации массивов | 70 |
| Проверка границ | 71 |
| Переменные и параметры | 72 |
| Стек и куча | 72 |
| Определенное присваивание | 73 |
| Стандартные значения | 74 |
| Параметры | 74 |
| Объявление неявно типизированных локальных переменных с помощью var | 79 |
| Выражения и операции | 80 |
| Первичные выражения | 80 |
| Пустые выражения | 80 |
| Выражения присваивания | 81 |
| Приоритеты и ассоциативность операций | 81 |
| Таблица операций | 82 |
| Операции для работы со значениями null | 85 |
| Операция объединения с null | 85 |
| null-условная операция (C# 6) | 85 |
| Операторы | 86 |
| Операторы объявления | 86 |
| Операторы выражений | 87 |
| Операторы выбора | 88 |
| Операторы итераций | 90 |
| Операторы перехода | 92 |
| Смешанные операторы | 93 |
| Пространства имен | 94 |
| Директива using | 95 |
| Директива using static (C# 6) | 95 |
| Правила внутри пространств имен | 96 |
| Назначение псевдонимов типам и пространствам имен | 97 |
| Дополнительные возможности пространств имен | 98 |
| Глава 3. Создание типов в C# | 101 |
| Классы | 101 |
| Поля | 101 |
| Методы | 102 |
| Конструкторы экземпляров | 103 |
| Инициализаторы объектов | 105 |
| Ссылка this | 106 |
| Свойства | 107 |

| | |
|---|-----|
| Индексаторы | 109 |
| Константы | 110 |
| Статические конструкторы | 111 |
| Статические классы | 113 |
| Финализаторы | 113 |
| Частичные типы и методы | 113 |
| Операция nameof (C# 6) | 114 |
| Наследование | 115 |
| Полиморфизм | 115 |
| Приведение и ссылочные преобразования | 116 |
| Виртуальные функции-члены | 118 |
| Абстрактные классы и абстрактные члены | 119 |
| Соккрытие унаследованных членов | 119 |
| Запечатывание функций и классов | 120 |
| Ключевое слово base | 121 |
| Конструкторы и наследование | 121 |
| Перегрузка и распознавание | 122 |
| Тип object | 123 |
| Упаковка и распаковка | 124 |
| Статическая проверка типов и проверка типов во время выполнения | 125 |
| Метод GetType и операция typeof | 125 |
| Метод ToString | 126 |
| Список членов object | 126 |
| Структуры | 126 |
| Семантика конструирования структуры | 127 |
| Модификаторы доступа | 128 |
| Примеры | 128 |
| Дружественные сборки | 129 |
| Установление верхнего предела доступности | 129 |
| Ограничения, накладываемые на модификаторы доступа | 129 |
| Интерфейсы | 130 |
| Расширение интерфейса | 131 |
| Явная реализация членов интерфейса | 131 |
| Реализация виртуальных членов интерфейса | 132 |
| Повторная реализация члена интерфейса в подклассе | 132 |
| Интерфейсы и упаковка | 134 |
| Перечисления | 135 |
| Преобразования перечислений | 135 |
| Перечисления флагов | 136 |
| Операции над перечислениями | 137 |
| Проблемы безопасности типов | 137 |
| Вложенные типы | 138 |
| Обобщения | 139 |
| Обобщенные типы | 139 |
| Для чего предназначены обобщения | 140 |
| Обобщенные методы | 141 |
| Объявление параметров типа | 142 |
| Операция typeof и несвязанные обобщенные типы | 142 |

| | |
|---|------------|
| Обобщенное значение <code>default</code> | 143 |
| Ограничения обобщений | 143 |
| Создание подклассов для обобщенных типов | 144 |
| Самоссылающиеся объявления обобщений | 145 |
| Статические данные | 145 |
| Параметры типа и преобразования | 145 |
| Ковариантность | 146 |
| Контравариантность | 149 |
| Сравнение обобщений C# и шаблонов C++ | 150 |
| Глава 4. Дополнительные средства C# | 151 |
| Делегаты | 151 |
| Написание подключаемых методов с помощью делегатов | 152 |
| Групповые делегаты | 153 |
| Целевые методы экземпляра и целевые статические методы | 154 |
| Обобщенные типы делегатов | 155 |
| Делегаты <code>Func</code> и <code>Action</code> | 155 |
| Сравнение делегатов и интерфейсов | 156 |
| Совместимость делегатов | 157 |
| События | 159 |
| Стандартный шаблон событий | 161 |
| Средства доступа к событию | 164 |
| Модификаторы событий | 165 |
| Лямбда-выражения | 165 |
| Явное указание типов лямбда-параметров | 166 |
| Захватывание внешних переменных | 166 |
| Анонимные методы | 169 |
| Операторы <code>try</code> и исключения | 169 |
| Конструкция <code>catch</code> | 171 |
| Блок <code>finally</code> | 173 |
| Генерация исключений | 174 |
| Основные свойства класса <code>System.Exception</code> | 176 |
| Общие типы исключений | 176 |
| Шаблон методов <code>TryXXX</code> | 177 |
| Альтернативы исключениям | 177 |
| Перечисление и итераторы | 178 |
| Перечисление | 178 |
| Инициализаторы коллекций | 179 |
| Итераторы | 179 |
| Семантика итератора | 180 |
| Компоновка последовательностей | 182 |
| Типы, допускающие значение <code>null</code> | 182 |
| Структура <code>Nullable<T></code> | 183 |
| Подъем операций | 184 |
| Тип <code>bool?</code> и операции <code>&</code> и <code> </code> | 186 |
| Типы, допускающие <code>null</code> , и операции для работы со значениями <code>null</code> | 186 |
| Сценарии использования типов, допускающих <code>null</code> | 187 |
| Альтернативы типам, допускающим значение <code>null</code> | 187 |

| | |
|---|------------|
| Перегрузка операций | 188 |
| Функции операций | 188 |
| Перегрузка операций эквивалентности и сравнения | 189 |
| Специальные неявные и явные преобразования | 190 |
| Перегрузка операций true и false | 190 |
| Расширяющие методы | 191 |
| Цепочки расширяющих методов | 192 |
| Неоднозначность и разрешение | 192 |
| Анонимные типы | 193 |
| Динамическое связывание | 195 |
| Сравнение статического и динамического связывания | 195 |
| Специальное связывание | 196 |
| Языковое связывание | 197 |
| Исключение <code>RuntimeBinderException</code> | 197 |
| Представление типа <code>dynamic</code> во время выполнения | 198 |
| Динамические преобразования | 198 |
| Сравнение <code>var</code> и <code>dynamic</code> | 199 |
| Динамические выражения | 199 |
| Динамические вызовы без динамических получателей | 200 |
| Статические типы в динамических выражениях | 201 |
| Невызываемые функции | 201 |
| Атрибуты | 202 |
| Классы атрибутов | 202 |
| Именованные и позиционные параметры атрибутов | 203 |
| Цели атрибутов | 203 |
| Указание нескольких атрибутов | 203 |
| Атрибуты информации о вызывающем компоненте | 204 |
| Небезопасный код и указатели | 205 |
| Основы указателей | 206 |
| Небезопасный код | 206 |
| Оператор <code>fixed</code> | 206 |
| Операция указателя на член | 207 |
| Массивы | 207 |
| <code>void*</code> | 208 |
| Указатели на неуправляемый код | 209 |
| Директивы препроцессора | 209 |
| Условные атрибуты | 209 |
| Директива <code>#pragma warning</code> | 210 |
| XML-документация | 211 |
| Стандартные XML-дескрипторы документации | 212 |
| Дескрипторы, определяемые пользователем | 213 |
| Перекрестные ссылки на типы или члены | 214 |
| Глава 5. Обзор .NET Framework | 215 |
| Среда CLR и ядро платформы | 218 |
| Системные типы | 218 |
| Обработка текста | 218 |
| Коллекции | 218 |
| Запросы | 218 |

| | |
|---|-----|
| XML | 219 |
| Диагностика и контракты кода | 219 |
| Параллелизм и асинхронность | 219 |
| Потоки данных и ввод-вывод | 219 |
| Работа с сетями | 220 |
| Сериализация | 220 |
| Сборки, рефлексия и атрибуты | 220 |
| Динамическое программирование | 221 |
| Безопасность | 221 |
| Расширенная многопоточность | 221 |
| Параллельное программирование | 221 |
| Домены приложений | 221 |
| Собственная возможность взаимодействия и возможность взаимодействия с COM | 222 |
| Прикладные технологии | 222 |
| Технологии пользовательских интерфейсов | 222 |
| Технологии серверной части | 225 |
| Технологии распределенных систем | 226 |
| Глава 6. Основы .NET Framework | 229 |
| Обработка строк и текста | 229 |
| Тип char | 229 |
| Тип string | 231 |
| Сравнение строк | 235 |
| Класс StringBuilder | 237 |
| Кодировка текста и Unicode | 238 |
| Дата и время | 242 |
| Структура TimeSpan | 242 |
| Структуры DateTime и DateTimeOffset | 243 |
| Даты и часовые пояса | 248 |
| DateTime и часовые пояса | 249 |
| DateTimeOffset и часовые пояса | 249 |
| TimeZone и TimeZoneInfo | 250 |
| Летнее время и DateTime | 253 |
| Форматирование и разбор | 255 |
| ToString и Parse | 255 |
| Поставщики форматов | 256 |
| Стандартные форматные строки и флаги разбора | 260 |
| Форматные строки для чисел | 260 |
| Перечисление NumberStyles | 263 |
| Форматные строки для даты/времени | 265 |
| Перечисление DateTimeStyles | 267 |
| Форматные строки для перечислений | 267 |
| Другие механизмы преобразования | 268 |
| Класс Convert | 268 |
| Класс XmlConvert | 270 |
| Преобразователи типов | 270 |
| Класс BitConverter | 271 |

| | |
|--|------------|
| Глобализация | 272 |
| Контрольный перечень глобализации | 272 |
| Тестирование | 272 |
| Работа с числами | 273 |
| Преобразования | 273 |
| Класс Math | 273 |
| Структура BigInteger | 274 |
| Структура Complex | 275 |
| Класс Random | 276 |
| Перечисления | 277 |
| Преобразования для перечислений | 277 |
| Перечисление значений enum | 279 |
| Как работают перечисления | 279 |
| Кортежи | 280 |
| Сравнение кортежей | 281 |
| Структура Guid | 281 |
| Сравнение эквивалентности | 282 |
| Эквивалентность значений и ссылочная эквивалентность | 282 |
| Стандартные протоколы эквивалентности | 283 |
| Эквивалентность и специальные типы | 287 |
| Сравнение порядка | 291 |
| Интерфейсы Comparable | 292 |
| Операции < и > | 293 |
| Реализация интерфейсов Comparable | 293 |
| Служебные классы | 294 |
| Класс Console | 294 |
| Класс Environment | 295 |
| Класс Process | 296 |
| Класс ApplicationContext | 297 |
| Глава 7. Коллекции | 299 |
| Перечисление | 299 |
| IEnumerable и IEnumerator | 300 |
| IEnumerable<T> и IEnumerator<T> | 301 |
| Реализация интерфейсов перечисления | 303 |
| Интерфейсы ICollection и IList | 306 |
| ICollection<T> и ICollection | 307 |
| IList<T> и IList | 308 |
| IReadOnlyList<T> | 309 |
| Класс Array | 310 |
| Конструирование и индексация | 312 |
| Перечисление | 314 |
| Длина и ранг | 314 |
| Поиск | 315 |
| Сортировка | 316 |
| Обращение порядка элементов | 317 |
| Копирование | 317 |
| Преобразование и изменение размера | 317 |

| | |
|---|------------|
| Списки, очереди, стеки и наборы | 318 |
| List<T> и ArrayList | 318 |
| LinkedList<T> | 321 |
| Queue<T> и Queue | 322 |
| Stack<T> и Stack | 323 |
| BitArray | 324 |
| HashSet<T> и SortedSet<T> | 324 |
| Словари | 326 |
| IDictionary<TKey, TValue> | 327 |
| IDictionary | 328 |
| Dictionary<TKey, TValue> и Hashtable | 328 |
| OrderedDictionary | 330 |
| ListDictionary и HybridDictionary | 330 |
| Отсортированные словари | 331 |
| Настраиваемые коллекции и прокси | 332 |
| Collection<T> и CollectionBase | 333 |
| KeyedCollection<TKey, TItem> и DictionaryBase | 335 |
| ReadOnlyCollection<T> | 337 |
| Подключение протоколов эквивалентности и порядка | 338 |
| IEqualityComparer и EqualityComparer | 339 |
| IComparer и Comparer | 341 |
| StringComparer | 342 |
| IStructuralEquatable и IStructuralComparable | 343 |
| Глава 8. Запросы LINQ | 345 |
| Начало работы | 345 |
| Текущий синтаксис | 347 |
| Выстраивание в цепочки операций запросов | 347 |
| Составление лямбда-выражений | 350 |
| Естественный порядок | 352 |
| Другие операции | 352 |
| Выражения запросов | 353 |
| Переменные диапазона | 355 |
| Сравнение синтаксиса запросов и синтаксиса SQL | 356 |
| Сравнение синтаксиса запросов и текущего синтаксиса | 356 |
| Запросы со смешанным синтаксисом | 357 |
| Отложенное выполнение | 357 |
| Повторная оценка | 358 |
| Захваченные переменные | 359 |
| Как работает отложенное выполнение | 360 |
| Построение цепочки декораторов | 361 |
| Каким образом выполняются запросы | 362 |
| Подзапросы | 363 |
| Подзапросы и отложенное выполнение | 366 |
| Стратегии композиции | 366 |
| Постепенное построение запросов | 366 |
| Ключевое слово into | 368 |
| Упаковка запросов | 369 |

| | |
|---|-----|
| Стратегии проекции | 370 |
| Инициализаторы объектов | 370 |
| Анонимные типы | 370 |
| Ключевое слово <code>let</code> | 371 |
| Интерпретируемые запросы | 372 |
| Каким образом работают интерпретируемые запросы | 374 |
| Комбинирование интерпретируемых и локальных запросов | 376 |
| <code>AsEnumerable</code> | 377 |
| LINQ to SQL и Entity Framework | 378 |
| Сущностные классы LINQ to SQL | 379 |
| Сущностные классы Entity Framework | 380 |
| <code>DataContext</code> и <code>ObjectContext</code> | 381 |
| Ассоциации | 385 |
| Отложенное выполнение в L2S и EF | 386 |
| <code>DataLoadOptions</code> | 387 |
| Энергичная загрузка в Entity Framework | 389 |
| Обновления | 389 |
| Отличия между API-интерфейсами L2S и EF | 391 |
| Построение выражений запросов | 392 |
| Сравнение делегатов и деревьев выражений | 392 |
| Деревья выражений | 394 |
| Глава 9. Операции LINQ | 397 |
| Обзор | 398 |
| Последовательность→последовательность | 399 |
| Последовательность→элемент или значение | 400 |
| Ничего→последовательность | 401 |
| Выполнение фильтрации | 401 |
| <code>Where</code> | 402 |
| <code>Take</code> и <code>Skip</code> | 403 |
| <code>TakeWhile</code> и <code>SkipWhile</code> | 404 |
| <code>Distinct</code> | 404 |
| Выполнение проекции | 404 |
| <code>Select</code> | 405 |
| <code>SelectMany</code> | 409 |
| Выполнение соединения | 416 |
| <code>Join</code> и <code>GroupJoin</code> | 416 |
| Операция <code>Zip</code> | 424 |
| Упорядочение | 424 |
| <code>OrderBy</code> , <code>OrderByDescending</code> , <code>ThenBy</code> и <code>ThenByDescending</code> | 424 |
| Группирование | 427 |
| <code>GroupBy</code> | 427 |
| Операции над множествами | 430 |
| <code>Concat</code> и <code>Union</code> | 430 |
| <code>Intersect</code> и <code>Except</code> | 431 |
| Методы преобразования | 431 |
| <code>OfType</code> и <code>Cast</code> | 431 |
| <code>ToArray</code> , <code>ToList</code> , <code>ToDictionary</code> и <code>ToLookup</code> | 433 |
| <code>AsEnumerable</code> и <code>AsQueryable</code> | 433 |

| | |
|--|-----|
| Операции над элементами | 434 |
| First, Last и Single | 434 |
| ElementAt | 435 |
| DefaultIfEmpty | 435 |
| Методы агрегирования | 436 |
| Count и LongCount | 436 |
| Min и Max | 436 |
| Sum и Average | 437 |
| Aggregate | 438 |
| Квантификаторы | 440 |
| Contains и Any | 440 |
| All и SequenceEqual | 441 |
| Методы генерации | 441 |
| Empty | 441 |
| Range и Repeat | 442 |
| Глава 10. LINQ to XML | 443 |
| Обзор архитектуры | 443 |
| Что собой представляет DOM-модель? | 443 |
| DOM-модель LINQ to XML | 444 |
| Обзор модели X-DOM | 444 |
| Загрузка и разбор | 446 |
| Сохранение и сериализация | 447 |
| Создание экземпляра X-DOM | 447 |
| Функциональное построение | 448 |
| Указание содержимого | 448 |
| Автоматическое глубокое копирование | 449 |
| Навигация и запросы | 450 |
| Навигация по дочерним узлам | 450 |
| Навигация по родительским узлам | 453 |
| Навигация по равноправным узлам | 453 |
| Навигация по атрибутам | 454 |
| Обновление модели X-DOM | 454 |
| Обновление простых значений | 455 |
| Обновление дочерних узлов и атрибутов | 455 |
| Обновление через родительский элемент | 456 |
| Работа со значениями | 457 |
| Установка значений | 457 |
| Получение значений | 458 |
| Значения и узлы со смешанным содержимым | 459 |
| Автоматическая конкатенация XText | 459 |
| Документы и объявления | 460 |
| XDocument | 460 |
| Объявления XML | 461 |
| Имена и пространства имен | 463 |
| Пространства имен в XML | 463 |
| Указание пространств имен в X-DOM | 465 |
| Модель X-DOM и стандартные пространства имен | 466 |
| Префиксы | 467 |

| | |
|---|-----|
| Аннотации | 468 |
| Проецирование в дерево X-DOM | 469 |
| Устранение пустых элементов | 471 |
| Потоковая передача проекции | 472 |
| Трансформирование X-DOM | 472 |
| Глава 11. Другие технологии XML | 475 |
| XmlReader | 476 |
| Чтение узлов | 477 |
| Чтение элементов | 479 |
| Чтение атрибутов | 482 |
| Пространства имен и префиксы | 483 |
| XmlWriter | 484 |
| Запись атрибутов | 485 |
| Запись других типов узлов | 485 |
| Пространства имен и префиксы | 486 |
| Шаблоны для использования XmlReader/XmlWriter | 486 |
| Работа с иерархическими данными | 486 |
| Смешивание XmlReader/XmlWriter с моделью X-DOM | 488 |
| XSD и проверка достоверности схемы | 490 |
| Выполнение проверки достоверности схемы | 491 |
| XSLT | 493 |
| Глава 12. Освобождение и сборка мусора | 495 |
| IDisposable, Dispose и Close | 495 |
| Стандартная семантика освобождения | 496 |
| Когда выполнять освобождение | 497 |
| Подключаемое освобождение | 499 |
| Очистка полей при освобождении | 500 |
| Автоматическая сборка мусора | 501 |
| Корневые объекты | 502 |
| Сборка мусора и WinRT | 503 |
| Финализаторы | 503 |
| Вызов метода Dispose из финализатора | 504 |
| Восстановление | 505 |
| Как работает сборщик мусора? | 507 |
| Технологии оптимизации | 508 |
| Принудительный запуск сборки мусора | 510 |
| Настройка сборки мусора | 511 |
| Нагрузка на память | 511 |
| Утечки управляемой памяти | 512 |
| Таймеры | 513 |
| Диагностика утечек памяти | 514 |
| Слабые ссылки | 515 |
| Слабые ссылки и кеширование | 516 |
| Слабые ссылки и события | 516 |
| Глава 13. Диагностика и контракты кода | 519 |
| Условная компиляция | 519 |
| Сравнение условной компиляции и статических переменных-флагов | 520 |

| | |
|--|-----|
| Атрибут <code>Conditional</code> | 521 |
| Классы <code>Debug</code> и <code>Trace</code> | 522 |
| <code>Fail</code> и <code>Assert</code> | 523 |
| <code>TraceListener</code> | 524 |
| Сброс и закрытие прослушивателей | 525 |
| Обзор контрактов кода | 526 |
| Зачем использовать контракты кода? | 527 |
| Принципы, лежащие в основе контрактов | 528 |
| Предусловия | 530 |
| <code>Contract.Requires</code> | 530 |
| <code>Contract.Requires<TException></code> | 532 |
| <code>Contract.EndContractBlock</code> | 533 |
| Предусловия и переопределенные методы | 534 |
| Постусловия | 534 |
| <code>Contract.Ensures</code> | 534 |
| <code>Contract.EnsuresOnThrow<TException></code> | 535 |
| <code>Contract.Result<T></code> и <code>Contract.ValueAtReturn<T></code> | 535 |
| <code>Contract.OldValue<T></code> | 536 |
| Постусловия и переопределенные методы | 536 |
| Утверждения и инварианты объектов | 536 |
| Утверждения | 536 |
| Инварианты объектов | 537 |
| Контракты на интерфейсах и абстрактных методах | 538 |
| Обработка нарушения контракта | 539 |
| Событие <code>ContractFailed</code> | 540 |
| Исключения внутри условий контракта | 541 |
| Избирательное применение контрактов | 541 |
| Контракты в окончательных сборках | 541 |
| Проверка на стороне вызывающего компонента | 542 |
| Статическая проверка контрактов | 542 |
| Атрибут <code>ContractVerification</code> | 543 |
| Базовые уровни | 544 |
| Атрибут <code>SuppressMessage</code> | 544 |
| Интеграция с отладчиком | 544 |
| Присоединение и останов | 544 |
| Атрибуты отладчика | 545 |
| Процессы и потоки процессов | 545 |
| Исследование выполняющихся процессов | 545 |
| Исследование потоков в процессе | 546 |
| <code>StackTrace</code> и <code>StackFrame</code> | 546 |
| Журналы событий <code>Windows</code> | 548 |
| Запись в журнал событий | 549 |
| Чтение журнала событий | 549 |
| Мониторинг журнала событий | 550 |
| Счетчики производительности | 550 |
| Перечисление доступных счетчиков производительности | 551 |
| Чтение данных счетчика производительности | 552 |
| Создание счетчиков и запись данных о производительности | 553 |
| Класс <code>Stopwatch</code> | 555 |

| | |
|--|-----|
| Глава 14. Параллелизм и асинхронность | 557 |
| Введение | 557 |
| Многопоточная обработка | 558 |
| Создание потока | 558 |
| Join и Sleep | 560 |
| Блокировка | 560 |
| Локальное или разделяемое состояние | 562 |
| Блокировка и безопасность потоков | 564 |
| Передача данных потоку | 565 |
| Обработка исключений | 566 |
| Потоки переднего плана или фоновые потоки | 568 |
| Приоритет потока | 569 |
| Передача сигналов | 569 |
| Многопоточность в обогащенных клиентских приложениях | 570 |
| Контексты синхронизации | 571 |
| Пул потоков | 572 |
| Задачи | 574 |
| Запуск задачи | 575 |
| Возвращение значений | 576 |
| Исключения | 577 |
| Продолжение | 578 |
| TaskCompletionSource | 580 |
| Task.Delay | 582 |
| Принципы асинхронности | 582 |
| Сравнение синхронных и асинхронных операций | 582 |
| Что собой представляет асинхронное программирование? | 583 |
| Асинхронное программирование и продолжение | 584 |
| Важность языковой поддержки | 585 |
| Асинхронные функции в C# | 587 |
| Ожидание | 587 |
| Написание асинхронных функций | 593 |
| Асинхронные лямбда-выражения | 597 |
| Асинхронные методы в WinRT | 598 |
| Асинхронность и контексты синхронизации | 599 |
| Оптимизация | 600 |
| Асинхронные шаблоны | 602 |
| Отмена | 602 |
| Сообщение о ходе работ | 604 |
| Асинхронный шаблон, основанный на задачах | 606 |
| Комбинаторы задач | 607 |
| Устаревшие шаблоны | 610 |
| Модель асинхронного программирования | 610 |
| Асинхронный шаблон на основе событий | 611 |
| BackgroundWorker | 612 |
| Глава 15. Потоки данных и ввод-вывод | 613 |
| Потоковая архитектура | 613 |
| Использование потоков | 615 |

| | |
|---|------------|
| Чтение и запись | 617 |
| Поиск | 618 |
| Заккрытие и сбрасывание | 618 |
| Тайм-ауты | 618 |
| Безопасность в отношении потоков управления | 619 |
| Потоки с опорными хранилищами | 619 |
| FileStream | 619 |
| MemoryStream | 623 |
| PipeStream | 623 |
| BufferedStream | 627 |
| Адаптеры потоков | 628 |
| Текстовые адаптеры | 628 |
| Двоичные адаптеры | 633 |
| Заккрытие и освобождение адаптеров потоков | 634 |
| Потоки со сжатием | 635 |
| Сжатие в памяти | 636 |
| Работа с zip-файлами | 637 |
| Операции с файлами и каталогами | 638 |
| Класс File | 638 |
| Класс Directory | 641 |
| FileInfo и DirectoryInfo | 642 |
| Path | 643 |
| Специальные папки | 644 |
| Запрашивание информации о томе | 646 |
| Перехват событий файловой системы | 647 |
| Файловый ввод-вывод в Windows Runtime | 648 |
| Работа с каталогами | 648 |
| Работа с файлами | 649 |
| Изолированное хранилище в приложениях Windows Store | 650 |
| Размещенные в памяти файлы | 650 |
| Размещенные в памяти файлы и произвольный файловый ввод-вывод | 650 |
| Размещенные в памяти файлы и разделяемая память | 651 |
| Работа с аксессуарами представлений | 652 |
| Изолированное хранилище | 653 |
| Типы изоляции | 653 |
| Чтение и запись в изолированное хранилище | 655 |
| Местоположение хранилища | 656 |
| Перечисление изолированного хранилища | 657 |
| Глава 16. Взаимодействие с сетью | 659 |
| Сетевая архитектура | 659 |
| Адреса и порты | 662 |
| Идентификаторы URI | 663 |
| Классы клиентской стороны | 665 |
| WebClient | 666 |
| WebRequest и WebResponse | 667 |
| HttpClient | 669 |
| Прокси-серверы | 673 |

| | |
|---|-----|
| Аутентификация | 674 |
| Обработка исключений | 676 |
| Работа с протоколом HTTP | 678 |
| Заголовки | 678 |
| Строки запросов | 678 |
| Выгрузка данных формы | 679 |
| Cookie-наборы | 680 |
| Аутентификация на основе форм | 681 |
| SSL | 683 |
| Реализация HTTP-сервера | 683 |
| Использование FTP | 686 |
| Использование DNS | 688 |
| Отправка сообщений электронной почты с помощью Smtplib | 688 |
| Использование TCP | 689 |
| Параллелизм и TCP | 692 |
| Получение почты POP3 с помощью TCP | 693 |
| TCP в Windows Runtime | 695 |
| Глава 17. Сериализация | 697 |
| Концепции сериализации | 697 |
| Механизмы сериализации | 697 |
| Форматеры | 700 |
| Сравнение явной и неявной сериализации | 700 |
| Сериализатор контрактов данных | 701 |
| СравнениеDataContractSerializer и NetDataContractSerializer | 701 |
| Использование сериализаторов | 702 |
| Сериализация подклассов | 704 |
| Объектные ссылки | 706 |
| Переносимость версий | 708 |
| Упорядочение членов | 709 |
| Пустые значения и null | 709 |
| Контракты данных и коллекции | 710 |
| Элементы коллекции, являющиеся подклассами | 711 |
| Настройка имен коллекции и элементов | 711 |
| Расширение контрактов данных | 712 |
| Ловушки сериализации и десериализации | 713 |
| Возможность взаимодействия с помощью [Serializable] | 714 |
| Возможность взаимодействия с помощью IXmlSerializable | 716 |
| Двоичный сериализатор | 716 |
| Начало работы | 716 |
| Атрибуты двоичной сериализации | 718 |
| [NonSerialized] | 718 |
| [OnDeserializing] и [OnDeserialized] | 718 |
| [OnSerializing] и [OnSerialized] | 719 |
| [OptionalField] и поддержка версий | 720 |
| Двоичная сериализация с помощью ISerializable | 721 |
| Создание подклассов из сериализируемых классов | 723 |

| | |
|--|------------|
| Сериализация XML | 724 |
| Начало работы с сериализацией на основе атрибутов | 724 |
| Подклассы и дочерние объекты | 726 |
| Сериализация коллекций | 729 |
| IXmlSerializable | 731 |
| Глава 18. Сборки | 733 |
| Содержимое сборки | 733 |
| Манифест сборки | 734 |
| Манифест приложения | 735 |
| Модули | 736 |
| Класс Assembly | 737 |
| Строгие имена и подписание сборок | 738 |
| Назначение сборке строгого имени | 739 |
| Отложенное подписание | 739 |
| Имена сборок | 741 |
| Полностью заданные имена | 741 |
| Класс AssemblyName | 742 |
| Информационная и файловая версии сборки | 742 |
| Подпись Authenticode | 743 |
| Подписание с помощью системы Authenticode | 744 |
| Проверка достоверности подписей Authenticode | 746 |
| Глобальный кеш сборок | 747 |
| Установка сборок в GAC | 748 |
| GAC и поддержка версий | 748 |
| Ресурсы и подчиненные сборки | 749 |
| Встраивание ресурсов напрямую | 750 |
| Файлы .resources | 751 |
| Файлы .resx | 752 |
| Подчиненные сборки | 754 |
| Культуры и подкультуры | 756 |
| Распознавание и загрузка сборок | 757 |
| Правила распознавания сборок и типов | 758 |
| Событие AssemblyResolve | 758 |
| Загрузка сборок | 759 |
| Развертывание сборок за пределами базовой папки | 762 |
| Упаковка однофайловой исполняемой сборки | 763 |
| Избирательное исправление | 765 |
| Работа со сборками, не имеющими ссылок на этапе компиляции | 765 |
| Глава 19. Рефлексия и метаданные | 767 |
| Рефлексия и активизация типов | 768 |
| Получение экземпляра Type | 768 |
| Имена типов | 770 |
| Базовые типы и интерфейсы | 771 |
| Создание экземпляров типов | 772 |
| Обобщенные типы | 773 |

| | |
|--|-----|
| Рефлексия и вызов членов | 774 |
| Типы членов | 776 |
| Сравнение членов C# и членов CLR | 778 |
| Члены обобщенных типов | 779 |
| Динамический вызов члена | 779 |
| Параметры методов | 780 |
| Использование делегатов для повышения производительности | 782 |
| Доступ к неоткрытым членам | 782 |
| Обобщенные методы | 784 |
| Анонимный вызов членов обобщенного интерфейса | 784 |
| Рефлексия сборок | 786 |
| Загрузка сборки в контекст, предназначенный только для рефлексии | 787 |
| Модули | 787 |
| Работа с атрибутами | 787 |
| Основы атрибутов | 788 |
| Атрибут <code>AttributeUsage</code> | 789 |
| Определение собственного атрибута | 790 |
| Извлечение атрибутов во время выполнения | 791 |
| Извлечение атрибутов в контексте, предназначенном только для рефлексии | 792 |
| Динамическая генерация кода | 793 |
| Генерация кода IL с помощью класса <code>DynamicMethod</code> | 793 |
| Стек оценки | 795 |
| Передача аргументов динамическому методу | 796 |
| Генерация локальных переменных | 796 |
| Ветвление | 797 |
| Создание объектов и вызов методов экземпляра | 798 |
| Обработка исключений | 799 |
| Выпускборок и типов | 800 |
| Сохранение сгенерированныхборок | 801 |
| Объектная модель <code>Reflection.Emit</code> | 802 |
| Выпуск членов типа | 803 |
| Выпуск методов | 803 |
| Выпуск полей и свойств | 805 |
| Выпуск конструкторов | 807 |
| Присоединение атрибутов | 808 |
| Выпуск обобщенных методов и типов | 808 |
| Определение обобщенных методов | 809 |
| Определение обобщенных типов | 810 |
| Сложности, связанные с генерацией | 810 |
| Несозданные закрытые обобщения | 810 |
| Циклические зависимости | 811 |
| Синтаксический разбор IL | 813 |
| Написание дизассемблера | 814 |
| Глава 20. Динамическое программирование | 819 |
| Исполняющая среда динамического языка | 819 |
| Унификация числовых типов | 821 |

| | |
|--|------------|
| Динамическое распознавание перегруженных членов | 822 |
| Упрощение шаблона Посетитель | 822 |
| Анонимный вызов членов обобщенного типа | 826 |
| Реализация динамических объектов | 828 |
| DynamicObject | 828 |
| ExpandableObject | 830 |
| Взаимодействие с динамическими языками | 831 |
| Передача состояния между C# и сценарием | 832 |
| Глава 21. Безопасность | 833 |
| Разрешения | 833 |
| CodeAccessPermission и PrincipalPermission | 834 |
| PermissionSet | 836 |
| Сравнение декларативной и императивной безопасности | 836 |
| Безопасность доступа кода | 837 |
| Применение безопасности доступа кода | 839 |
| Проверка на полное доверие | 840 |
| Разрешение вызывающих компонентов с частичным доверием | 840 |
| Повышение привилегий | 840 |
| APTCSA и [SecurityTransparent] | 841 |
| Модель прозрачности | 842 |
| Работа модели прозрачности | 843 |
| Как создавать библиотеки APTCSA с применением прозрачности | 846 |
| Прозрачность в сценариях с полным доверием | 849 |
| Помещение в песочницу другой сборки | 851 |
| Утверждение разрешений | 852 |
| Подсистема безопасности операционной системы | 854 |
| Выполнение от имени учетной записи стандартного пользователя | 855 |
| Повышение полномочий до административных и виртуализация | 856 |
| Безопасность на основе удостоверений и ролей | 857 |
| Назначение пользователей и ролей | 857 |
| Обзор криптографии | 858 |
| Защита данных Windows | 858 |
| Хеширование | 860 |
| Симметричное шифрование | 861 |
| Шифрование в памяти | 863 |
| Соединение в цепочку потоков шифрования | 864 |
| Освобождение объектов шифрования | 865 |
| Управление ключами | 866 |
| Шифрование с открытым ключом и подписание | 866 |
| Класс RSA | 867 |
| Цифровые подписи | 868 |
| Глава 22. Расширенная многопоточность | 871 |
| Обзор синхронизации | 872 |
| Монопольное блокирование | 872 |
| Оператор lock | 873 |
| Monitor.Enter и Monitor.Exit | 874 |
| Выбор объекта синхронизации | 875 |

| | |
|--|------------|
| Когда нужна блокировка | 875 |
| Блокирование и атомарность | 876 |
| Вложенное блокирование | 877 |
| Взаимоблокировки | 878 |
| Производительность | 879 |
| Mutex | 879 |
| Блокирование и безопасность к потокам | 880 |
| Безопасность к потокам и типы .NET Framework | 882 |
| Безопасность к потокам в серверах приложений | 884 |
| Неизменяемые объекты | 885 |
| Немонопольное блокирование | 886 |
| Семафор | 886 |
| Блокировки объектов чтения/записи | 887 |
| Сигнализирование с помощью дескрипторов ожидания событий | 892 |
| AutoResetEvent | 892 |
| ManualResetEvent | 895 |
| CountdownEvent | 895 |
| Создание межпроцессного объекта EventWaitHandle | 896 |
| Дескрипторы ожидания и продолжение | 897 |
| Преобразование дескрипторов ожидания в задачи | 897 |
| WaitAny, WaitAll и SignalAndWait | 898 |
| Класс Barrier | 899 |
| Ленивая инициализация | 901 |
| Lazy<T> | 902 |
| LazyInitializer | 902 |
| Локальное хранилище потока | 903 |
| [ThreadStatic] | 904 |
| ThreadLocal<T> | 904 |
| GetData и SetData | 905 |
| Interrupt и Abort | 905 |
| Suspend и Resume | 906 |
| Таймеры | 907 |
| Многопоточные таймеры | 908 |
| Однопоточные таймеры | 910 |
| Глава 23. Параллельное программирование | 911 |
| Для чего нужна инфраструктура PFX | 911 |
| Концепции PFX | 912 |
| Компоненты PFX | 912 |
| Когда необходимо использовать инфраструктуру PFX | 914 |
| PLINQ | 914 |
| Продвижение параллельного выполнения | 917 |
| PLINQ и упорядочивание | 917 |
| Ограничения PLINQ | 918 |
| Пример: параллельная программа проверки орфографии | 918 |
| Функциональная чистота | 920 |
| Установка степени параллелизма | 921 |
| Отмена | 922 |
| Оптимизация PLINQ | 922 |

| | |
|--|-----|
| Класс Parallel | 928 |
| Parallel.Invoke | 928 |
| Parallel.For и Parallel.ForEach | 929 |
| Параллелизм задач | 934 |
| Создание и запуск задач | 935 |
| Ожидание на множестве задач | 936 |
| Отмена задач | 937 |
| Продолжение | 938 |
| Планировщики задач | 942 |
| TaskFactory | 942 |
| Работа с AggregateException | 943 |
| Flatten и Handle | 944 |
| Параллельные коллекции | 945 |
| IProducerConsumerCollection<T> | 946 |
| ConcurrentBag<T> | 947 |
| BlockingCollection<T> | 948 |
| Реализация очереди производителей/потребителей | 949 |
| Глава 24. Домены приложений | 953 |
| Архитектура доменов приложений | 953 |
| Создание и уничтожение доменов приложений | 954 |
| Использование нескольких доменов приложений | 956 |
| Использование DoCallback | 958 |
| Мониторинг доменов приложений | 958 |
| Домены и потоки | 959 |
| Разделение данных между доменами | 960 |
| Разделение данных через ячейки | 960 |
| Использование Remoting внутри процесса | 961 |
| Изолирование типов и сборок | 963 |
| Глава 25. Способность к взаимодействию | 967 |
| Обращение к низкоуровневым DLL-библиотекам | 967 |
| Маршализация типов | 968 |
| Маршализация общих типов | 968 |
| Маршализация классов и структур | 969 |
| Маршализация параметров in и out | 970 |
| Обратные вызовы из неуправляемого кода | 971 |
| Эмуляция объединения C | 971 |
| Разделяемая память | 972 |
| Отображение структуры на неуправляемую память | 975 |
| fixed и fixed { . . . } | 977 |
| Взаимодействие с COM | 979 |
| Назначение COM | 979 |
| Основы системы типов COM | 979 |
| Обращение к компоненту COM из C# | 980 |
| Необязательные параметры и именованные аргументы | 982 |
| Неявные параметры ref | 982 |
| Индексаторы | 982 |
| Динамическое связывание | 983 |

| | |
|---|-------------|
| Внедрение типов взаимодействия | 984 |
| Эквивалентность типов | 984 |
| Основные сборки взаимодействия | 985 |
| Открытие объектов C# для COM | 985 |
| Глава 26. Регулярные выражения | 987 |
| Основы регулярных выражений | 987 |
| Скомпилированные регулярные выражения | 989 |
| RegexOptions | 989 |
| Отмена символов | 989 |
| Наборы символов | 991 |
| Квантификаторы | 992 |
| Жадные и ленивые квантификаторы | 992 |
| Утверждения нулевой ширины | 993 |
| Просмотр вперед и просмотр назад | 993 |
| Привязки | 994 |
| Границы слов | 995 |
| Группы | 995 |
| Именованные группы | 996 |
| Замена и разделение текста | 997 |
| Делегат MatchEvaluator | 997 |
| Разделение текста | 998 |
| Рецептурный справочник по регулярным выражениям | 998 |
| Рецепты | 998 |
| Справочник по языку регулярных выражений | 1001 |
| Глава 27. Компилятор Roslyn | 1005 |
| Архитектура Roslyn | 1006 |
| Рабочие области | 1006 |
| Синтаксические деревья | 1006 |
| Структура SyntaxTree | 1007 |
| Получение синтаксического дерева | 1010 |
| Обход и поиск в дереве | 1011 |
| Трансформация синтаксического дерева | 1018 |
| Объекты компиляции и семантические модели | 1022 |
| Создание объекта компиляции | 1022 |
| Выпуск сборки | 1023 |
| Выдача запросов к семантической модели | 1023 |
| Пример: переименование символа | 1028 |
| Предметный указатель | 1032 |



Другие технологии XML

Пространство имен `System.Xml` содержит следующие пространства имен и основные классы:

System.Xml.*

XmlReader и XmlWriter

Высокопроизводительные однонаправленные курсоры для чтения и записи в XML-поток.

XmlDocument

Представляет XML-документ в DOM-модели стиля W3C (устарел).

System.Xml.XPath

Инфраструктура и API-интерфейс (`XPathNavigator`) для XPath — основанного на строках языка для написания запросов XML.

System.Xml.Linq

Современная DOM-модель, ориентированная на LINQ, которая предназначена для работы с XML.

System.Xml.Schema

Инфраструктура и API-интерфейс для схем XSD (W3C).

System.Xml.Xsl

Инфраструктура и API-интерфейс (`XslCompiledTransform`) для выполнения трансформаций XSLT структур XML (W3C).

System.Xml.Serialization

Поддерживает сериализацию классов классов в и из XML-данных (см. главу 17).

W3C — это аббревиатура, обозначающая консорциум World Wide Web Consortium, где определяются стандарты XML.

Статический класс `XmlConvert`, предназначенный для разбора и форматирования XML-строк, рассматривался в главе 6.

XmlReader

`XmlReader` – это высокопроизводительный класс для чтения XML-потока низкоуровневым однонаправленным способом.

Взгляните на следующий XML-файл:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Для создания экземпляра `XmlReader` вызывается статический метод `XmlReader.Create`, которому передается объект `Stream`, `TextReader` или строка URI. Например:

```
using (XmlReader reader = XmlReader.Create ("customer.xml"))
    ...
```



Поскольку класс `XmlReader` позволяет читать из потенциально медленных источников (`Stream` и `URI`), он предлагает асинхронные версии большинства своих методов, так что можно легко писать неблокирующий код. Асинхронность подробно обсуждается в главе 14.

Ниже показано, как сконструировать экземпляр `XmlReader`, который читает из строки:

```
XmlReader reader = XmlReader.Create (
    new System.IO.StringReader (myString));
```

Для управления настройками разбора и проверки достоверности можно также передавать объект `XmlReaderSettings`. В частности, следующие три свойства `XmlReaderSettings` полезны для пропуска избыточного содержимого:

```
bool IgnoreComments           // Пропускать узлы комментариев?
bool IgnoreProcessingInstructions // Пропускать инструкции обработки?
bool IgnoreWhitespace         // Пропускать пробельные символы?
```

В приведенном далее примере средству чтения сообщается о том, что узлы с пробельными символами, которые отвлекают внимание в типовых сценариях, выдаваться не должны:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
using (XmlReader reader = XmlReader.Create ("customer.xml", settings))
    ...
```

Еще одним полезным свойством `XmlReaderSettings` является `ConformanceLevel`. Его стандартное значение `Document` указывает средству чтения на то, что необходимо предполагать наличие допустимого XML-документа с единственным корневым узлом. Такая проблема возникает, когда нужно прочитать только внутреннюю порцию XML, содержащую несколько узлов:

```
<firstname>Jim</firstname>
<lastname>Bo</lastname>
```

Чтобы прочитать это без генерации исключения, потребуется установить `ConformanceLevel` в `Fragment`.

Класс `XmlReaderSettings` также имеет свойство по имени `CloseInput`, которое указывает на то, должен ли закрываться лежащий в основе поток, когда закрывается средство чтения (в `XmlWriterSettings` существует аналогичное свойство под названием `CloseOutput`). Стандартное значение для свойств `CloseInput` и `CloseOutput` равно `false`.

Чтение узлов

Единицами XML-потока являются *узлы XML*. Средство чтения перемещается по потоку в текстовом порядке (сначала в глубину). Свойство `Depth` средства чтения возвращает текущую глубину курсора.

Наиболее простой способ чтения из `XmlReader` предполагает вызов метода `Read`. Он осуществляет перемещение на следующий узел в XML-потоке подобно методу `MoveNext` в интерфейсе `IEnumerator`. Первый вызов `Read` устанавливает курсор на первый узел. Когда метод `Read` возвращает `false`, это означает, что курсор переместился за последний узел, и в данном случае экземпляр `XmlReader` должен быть закрыт и освобожден.

В следующем примере мы читаем каждый узел в XML-потоке, выводя по мере продвижения тип узла:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using (XmlReader reader = XmlReader.Create ("customer.xml", settings))
    while (reader.Read())
    {
        Console.Write (new string (' ', reader.Depth*2)); // Вывести отступ
        Console.WriteLine (reader.NodeType);
    }
```

Ниже показан вывод:

```
XmlDeclaration
Element
  Element
    Text
  EndElement
Element
  Text
EndElement
EndElement
```



Атрибуты в обход на основе `Read` не включаются (см. раздел “Чтение атрибутов” далее в этой главе).

Свойство `NodeType` имеет тип `XmlNodeType`, который представляет собой перечисление со следующими членами:

| | | |
|-----------------------------|------------------------------------|------------------------------------|
| <code>None</code> | <code>Comment</code> | <code>Document</code> |
| <code>XmlDeclaration</code> | <code>Entity</code> | <code>DocumentType</code> |
| <code>Element</code> | <code>EndElement</code> | <code>DocumentFragment</code> |
| <code>EndElement</code> | <code>EntityReference</code> | <code>Notation</code> |
| <code>Text</code> | <code>ProcessingInstruction</code> | <code>Whitespace</code> |
| <code>Attribute</code> | <code>CDATA</code> | <code>SignificantWhitespace</code> |

Два строковых свойства в XmlReader — Name и Value — предоставляют доступ к содержимому узла. В зависимости от типа узла, наполняется либо Name, либо Value (или оба):

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
settings.DtdProcessing = DtdProcessing.Parse; // Требуется для чтения DTD
using (XmlReader r = XmlReader.Create ("customer.xml", settings))
    while (r.Read())
    {
        Console.Write (r.NodeType.ToString().PadRight (17, '-'));
        Console.Write ("> ".PadRight (r.Depth * 3));

        switch (r.NodeType)
        {
            case XmlNodeType.Element:
            case XmlNodeType.EndElement:
                Console.WriteLine (r.Name); break;

            case XmlNodeType.Text:
            case XmlNodeType.CDATA:
            case XmlNodeType.Comment:
            case XmlNodeType.XmlDeclaration:
                Console.WriteLine (r.Value); break;

            case XmlNodeType.DocumentType:
                Console.WriteLine (r.Name + " - " + r.Value); break;

            default: break;
        }
    }
}
```

Для демонстрации этого мы расширим наш XML-файл, включив тип документа, сущность, CDATA и комментарий:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE customer [ <!ENTITY tc "Top Customer" ]>
<customer id="123" status="archived">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
    <quote><![CDATA[C#'s operators include: < > &]]></quote>
    <notes>Jim Bo is a &tc;</notes>
    <!-- That wasn't so bad! -->
</customer>
```

Сущность подобна макросу; CDATA похоже на дословную строку (@"...") в C#. Ниже показан результат:

```
XmlDeclaration----> version="1.0" encoding="utf-8"
DocumentType-----> customer - <!ENTITY tc "Top Customer">
Element-----> customer
Element-----> firstname
Text-----> Jim
EndElement-----> firstname
Element-----> lastname
Text-----> Bo
EndElement-----> lastname
Element-----> quote
CDATA-----> C#'s operators include: < > &
```

```

EndElement-----> quote
Element-----> notes
Text-----> Jim Bo is a Top Customer
EndElement-----> notes
Comment-----> That wasn't so bad!
EndElement-----> customer

```

Класс `XmlReader` автоматически распознает сущности, так что в рассмотренном примере ссылка на сущность `&tc;` расширяется в `Top Customer`.

Чтение элементов

Часто структура читаемого XML-документа уже известна. Чтобы помочь в этом отношении, класс `XmlReader` предлагает набор методов, которые выполняют чтение, *предполагая* наличие определенной структуры. Они упрощают код и одновременно с этим выполняют некоторую проверку достоверности.



Класс `XmlReader` генерирует исключение `XmlException`, если любая проверка достоверности терпит неудачу. Класс `XmlException` имеет свойства `LineNumber` и `LinePosition`, которые указывают, где произошла ошибка — регистрация этой информации в журнале очень важна в случае крупных XML-файлов!

Метод `ReadStartElement` проверяет, что текущий `NodeType` является `Element`, и затем вызывает метод `Read`. Если указано имя, то он проверяет, что оно совпадает с именем текущего элемента.

Метод `ReadEndElement` удостоверяется в том, что текущий `NodeType` — это `EndElement`, и затем вызывает метод `Read`.

Например, мы могли бы прочитать этот узел:

```
<firstname>Jim</firstname>
```

следующим образом:

```

reader.ReadStartElement ("firstname");
Console.WriteLine (reader.Value);
reader.Read ();
reader.ReadEndElement ();

```

Метод `ReadElementContentAsString` делает все описанные ранее действия за раз. Он читает начальный элемент, текстовый узел и конечный элемент, возвращая содержимое в виде строки:

```
string firstName = reader.ReadElementContentAsString ("firstname", "");
```

Второй аргумент ссылается на пространство имен, которое в этом примере оставлено пустым. Доступны также типизированные версии данного метода, такие как `ReadElementContentAsInt`, разбирающие результат. Вернемся к нашему исходному XML-документу:

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <creditlimit>500.00</creditlimit> <!-- Да, мы вставили этот комментарий! -->
</customer>

```

Его можно прочитать следующим образом:

```

XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using (XmlReader r = XmlReader.Create ("customer.xml", settings))
{
    r.MoveToContent(); // Пропустить XML-объявление
    r.ReadStartElement ("customer");
    string firstName = r.ReadElementContentAsString ("firstname", "");
    string lastName = r.ReadElementContentAsString ("lastname", "");
    decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");

    r.MoveToContent(); // Пропустить этот надоедливый комментарий
    r.ReadEndElement(); // Читать закрывающий дескриптор customer
}

```



Метод `MoveToContent` чрезвычайно удобен. Он пропускает все малоинтересное: XML-объявления, пробельные символы, комментарии и инструкции обработки. Посредством свойств `XmlReaderSettings` можно заставить средство чтения делать большинство из этого автоматически.

Необязательные элементы

Предположим в предыдущем примере, что элемент `<lastname>` является необязательным. Решение очень простое:

```

r.ReadStartElement ("customer");
string firstName = r.ReadElementContentAsString ("firstname", "");
string lastName = r.Name == "lastname"
    ? r.ReadElementContentAsString() : null;
decimal creditLimit = r.ReadElementContentAsDecimal ("creditlimit", "");

```

Случайный порядок элементов

Примеры, приводимые в этом разделе, полагаются на то, что элементы в XML-файле расположены в установленном порядке. Чтобы справиться с элементами, представленными в другом порядке, проще всего прочитать их с помещением в дерево X-DOM. Мы покажем, как это делать, в разделе “Шаблоны для использования `XmlReader/XmlWriter`” далее в главе.

Пустые элементы

Способ, которым класс `XmlReader` обрабатывает пустые элементы, таит в себе серьезную ловушку. Рассмотрим следующий элемент:

```
<customerList></customerList>
```

Вот его эквивалент в XML:

```
<customerList/>
```

Тем не менее, `XmlReader` трактует их по-разному. В первом случае приведенный ниже код работает так, как было задумано:

```

reader.ReadStartElement ("customerList");
reader.ReadEndElement();

```

Во втором случае метод `ReadEndElement` генерирует исключение, т.к. отсутствует отдельный “конечный элемент”, на который рассчитывает класс `XmlReader`. Обходной путь предусматривает добавление проверки на предмет пустых элементов, как показано ниже:

```
bool isEmpty = reader.IsEmptyElement;
reader.ReadStartElement ("customerList");
if (!isEmpty) reader.ReadEndElement();
```

В действительности эта неприятность возникает, только когда рассматриваемый элемент может содержать дочерние элементы (скажем, список заказчиков). В случае элементов, которые содержат простой текст (вроде `firstname`), проблемы можно избежать путем вызова такого метода, как `ReadElementContentAsString`. Методы `ReadElementXXX` корректно обрабатывают оба вида пустых элементов.

Другие методы `ReadXXX`

В табл. 11.1 приведена сводка по всем методам `ReadXXX` в классе `XmlReader`. Большинство из них предназначено для работы с элементами. Выделенная полужирным часть в примере XML-фрагмента представляет собой раздел, который читается описываемым методом.

Таблица 11.1. Методы чтения

| Методы | Типы узлов, на которых методы работают | Пример XML-фрагмента | Входные параметры | Возвращаемые данные |
|--------------------------------------|--|---|-------------------|-----------------------------------|
| <code>ReadContentAsXXX</code> | Text | <code><a>x</code> | | x |
| <code>ReadString</code> | Text | <code><a>x</code> | | x |
| <code>ReadElementString</code> | Element | <code><a>x</code> | | x |
| <code>ReadElementContentAsXXX</code> | Element | <code><a>x</code> | | x |
| <code>ReadInnerXml</code> | Element | <code><a>x</code> | | x |
| <code>ReadOuterXml</code> | Element | <code><a>x</code> | | <code><a>x</code> |
| <code>ReadStartElement</code> | Element | <code><a>x</code> | | |
| <code>ReadEndElement</code> | Element | <code><a>x</code> | | |
| <code>ReadSubtree</code> | Element | <code><a>x</code> | | <code><a>x</code> |
| <code>ReadToDescendant</code> | Element | <code><a>x</code> | "b" | |
| <code>ReadToFollowing</code> | Element | <code><a>x</code> | "b" | |
| <code>ReadToNextSibling</code> | Element | <code><a>x</code> | "b" | |
| <code>ReadAttributeValue</code> | Attribute | См. раздел "Чтение атрибутов" далее в главе | | |

Методы `ReadContentAsXXX` разбирают текстовый узел в тип `XXX`. Внутренне класс `XmlConvert` выполняет преобразование из строки в этот тип. Текстовый узел может находиться внутри элемента или атрибута.

Методы `ReadElementContentAsXXX` – это оболочки вокруг соответствующих методов `ReadContentAsXXX`. Они применяются к узлу *элемента*, а не к *текстовому* узлу, заключенному в элемент.



Типизированные методы `ReadXXX` также имеют версии, которые читают в байтовый массив данные в форматах Base64 и BinHex.

Метод `ReadInnerXml` обычно применяется к элементу; он читает и возвращает элемент со всеми его потомками. В случае применения к атрибуту этот метод возвращает значение атрибута.

Метод `ReadOuterXml` аналогичен `ReadInnerXml` с тем лишь отличием, что он включает, а не исключает элемент в позиции курсора.

Метод `ReadSubtree` возвращает новый экземпляр `XmlReader`, который обеспечивает представление только текущего элемента (и его потомков). Чтобы исходный `XmlReader` мог безопасно продолжить чтение, этот экземпляр должен быть закрыт. В момент, когда новый экземпляр `XmlReader` закрывается, позиция курсора исходного `XmlReader` перемещается в конец поддерева.

Метод `ReadToDescendant` перемещает курсор в начало первого узла-потомка с указанным именем/пространством имен.

Метод `ReadToFollowing` перемещает курсор в начало первого узла — независимо от глубины — с указанным именем/пространством имен.

Метод `ReadToNextSibling` перемещает курсор в начало первого родственного узла с указанным именем/пространством имен.

Методы `ReadString` и `ReadElementString` ведут себя подобно `ReadContentAsString` и `ReadElementContentAsString`, но с тем отличием, что генерируют исключение, если внутри элемента обнаруживается более *одного* текстового узла. В общем случае использования этих методов следует избегать, потому что они генерируют исключение, если элемент содержит комментарий.

Чтение атрибутов

Класс `XmlReader` предоставляет индексатор, обеспечивающий прямой (произвольный) доступ к атрибутам элемента — по имени или по позиции. Применение индексатора эквивалентно вызову метода `GetAttribute`.

Имея следующий XML-фрагмент:

```
<customer id="123" status="archived"/>
```

мы могли бы прочесть его атрибуты так:

```
Console.WriteLine (reader ["id"]);           // 123
Console.WriteLine (reader ["status"]);       // archived
Console.WriteLine (reader ["bogus"] == null); // True
```



Для того чтобы читать атрибуты, экземпляр `XmlReader` должен быть позиционирован на начальный элемент. После вызова метода `ReadStartElement` атрибуты исчезают навсегда!

Хотя порядок атрибутов семантически несущественен, доступ к атрибутам возможен по их ординальным позициям. Предыдущий пример можно переписать следующим образом:

```
Console.WriteLine (reader [0]);           // 123
Console.WriteLine (reader [1]);           // archived
```

Индексатор также позволяет указывать пространство имен атрибута, если оно имеется.

Свойство `AttributeCount` возвращает количество атрибутов для текущего узла.

Узлы атрибутов

Для явного обхода узлов атрибутов потребуется сделать специальное ответвление от нормального пути, совершаемого простым вызовом метода `Read`. Хорошим поводом поступить так является необходимость разбора значений атрибутов в другие типы с помощью методов `ReadContentAsXXX`.

Ответвление должно начинаться с *начального элемента*. Для упрощения работы во время обхода атрибутов правило однонаправленности ослабляется: вызывая метод `MoveToAttribute`, можно переходить к любому атрибуту (вперед или назад).



Метод `MoveToElement` возвращает начальный элемент из любого места внутри ответвления узла атрибута.

Вернувшись к предыдущему примеру:

```
<customer id="123" status="archived"/>
```

можно поступить так:

```
reader.MoveToAttribute ("status");  
string status = reader.ReadContentAsString();  
reader.MoveToAttribute ("id");  
int id = reader.ReadContentAsInt();
```

Метод `MoveToAttribute` возвращает `false`, если указанный атрибут не существует.

Можно также выполнить обход всех атрибутов в последовательности, вызывая метод `MoveToFirstAttribute`, а затем метод `MoveToNextAttribute`:

```
if (reader.MoveToFirstAttribute())  
    do  
    {  
        Console.WriteLine (reader.Name + "=" + reader.Value);  
    }  
    while (reader.MoveToNextAttribute());  
// ВЫВОД:  
id=123  
status=archived
```

Пространства имен и префиксы

Класс `XmlReader` предлагает две параллельные системы для ссылки на имена элементов и атрибутов:

- `Name`
- `NamespaceURI` и `LocalName`

Всякий раз, когда вы читаете свойство `Name` элемента или вызываете метод, принимающий одиночный аргумент `name`, вы используете первую систему. Такой подход хорошо работает в отсутствие каких-либо пространств имен или префиксов; в противном случае это действует в грубой и буквальной манере. Пространства имен игнорируются, а префиксы включаются в точности так, как они записаны. Например:

| Пример фрагмента | Значение <code>Name</code> |
|--|----------------------------|
| <code><customer ...></code> | <code>customer</code> |
| <code><customer xmlns='blah' ...></code> | <code>customer</code> |
| <code><x:customer ...></code> | <code>x:customer</code> |

Приведенный ниже код работает с первыми двумя случаями:

```
reader.ReadStartElement ("customer");
```

Для обработки третьего случая требуется следующий код:

```
reader.ReadStartElement ("x:customer");
```

Вторая система работает через два свойства, *осведомленные о пространствах имен*: `NamespaceURI` и `LocalName`. Упомянутые свойства принимают во внимание префиксы и стандартные пространства имен, определенные родительскими элементами. Префиксы автоматически расширяются. Это означает, что свойство `NamespaceURI` всегда отражает семантически корректное пространство имен для текущего элемента, а свойство `LocalName` всегда свободно от префиксов.

При передаче двух аргументов имен в такой метод, как `ReadStartElement`, вы применяете ту же самую систему. Например, взгляните на следующий XML-фрагмент:

```
<customer xmlns="DefaultNamespace" xmlns:other="OtherNamespace">
  <address>
    <other:city>
      ...
    </other:city>
  </address>
</customer>
```

Прочитать его можно было бы так:

```
reader.ReadStartElement ("customer", "DefaultNamespace");
reader.ReadStartElement ("address", "DefaultNamespace");
reader.ReadStartElement ("city", "OtherNamespace");
```

Абстрагирование от префиксов обычно является именно тем, что нужно. При необходимости посредством свойства `Prefix` можно просмотреть, какой префикс использовался, и с помощью метода `LookupNamespace` преобразовать его в пространство имен.

XmlWriter

Класс `XmlWriter` — это однонаправленное средство записи в XML-поток. Проектное решение, положенное в основу `XmlWriter`, симметрично таковому в классе `XmlReader`.

Как и `XmlTextReader`, экземпляр `XmlWriter` конструируется вызовом метода `Create`, которому передается необязательный объект настроек. В приведенном ниже примере мы разрешаем отступы, чтобы сделать вывод удобным для восприятия человеком, и затем записываем в простой XML-файл:

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;

using (XmlWriter writer = XmlWriter.Create ("..\..\..\foo.xml", settings))
{
  writer.WriteStartElement ("customer");
  writer.WriteElementString ("firstname", "Jim");
  writer.WriteElementString ("lastname", "Bo");
  writer.WriteEndElement();
}
```

В результате получается следующий документ (тот же самый, что и в файле, который мы читали в первом примере применения класса `XmlReader`):

```
<?xml version="1.0" encoding="utf-8" ?>
<customer>
```

```
<firstname>Jim</firstname>
<lastname>Bo</lastname>
</customer>
```

Класс `XmlWriter` автоматически записывает объявление в начале, если только в `XmlWriterSettings` не указано обратное путем установки свойства `OmitXmlDeclaration` в `true` или свойства `ConformanceLevel` в `Fragment`. В последнем случае также разрешается запись нескольких корневых узлов — то, что иначе приводит к генерации исключения.

Метод `WriteValue` записывает одиночный текстовый узел. Он принимает строковые и нестроковые типы, такие как `bool` и `DateTime`, внутренне используя класс `XmlConvert` для выполнения совместимых с XML преобразований строк:

```
writer.WriteStartElement ("birthdate");
writer.WriteValue (DateTime.Now);
writer.WriteEndElement ();
```

В противоположность этому, если мы вызовем:

```
WriteElementString ("birthdate", DateTime.Now.ToString ());
```

то результат окажется несовместимым с XML и уязвимым к некорректному разбору. Вызов метода `WriteString` эквивалентен вызову метода `WriteValue` со строкой. Класс `XmlWriter` автоматически защищает символы, которые в противном случае были бы недопустимыми внутри атрибута либо элемента, такие как `&`, `<`, `>`, и расширенные символы `Unicode`.

Запись атрибутов

Атрибуты можно записывать немедленно после записи начального элемента:

```
writer.WriteStartElement ("customer");
writer.WriteAttributeString ("id", "1");
writer.WriteAttributeString ("status", "archived");
```

Для записи нестроковых значений вызывайте методы `WriteStartAttribute`, `WriteValue` и `WriteEndAttribute`.

Запись других типов узлов

В классе `XmlWriter` также определены следующие методы для записи других разновидностей узлов:

```
WriteBase64 // для двоичных данных
WriteBinHex // для двоичных данных
WriteCDATA
WriteComment
WriteDocType
WriteEntityRef
WriteProcessingInstruction
WriteRaw
WriteWhitespace
```

Метод `WriteRaw` внедряет строку прямо в выходной поток. Имеется также метод `WriteNode`, который принимает экземпляр `XmlReader` и копирует из него все данные.

Пространства имен и префиксы

Перегруженные версии методов `Write*` позволяют ассоциировать элемент или атрибут с пространством имен. Давайте перепишем содержимое XML-файла из предыдущего примера. На этот раз мы будем связывать все элементы с пространством имен `http://oreilly.com`, объявив префикс `o` в элементе `customer`:

```
writer.WriteStartElement ("o", "customer", "http://oreilly.com");
writer.WriteElementString ("o", "firstname", "http://oreilly.com", "Jim");
writer.WriteElementString ("o", "lastname", "http://oreilly.com", "Bo");
writer.WriteEndElement ();
```

Вывод теперь выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<o:customer xmlns:o='http://oreilly.com'>
  <o:firstname>Jim</o:firstname>
  <o:lastname>Bo</o:lastname>
</o:customer>
```

Обратите внимание, что для краткости класс `XmlWriter` опускает объявления пространств имен в дочерних элементах, если они уже объявлены их родительским элементом.

Шаблоны для использования `XmlReader/XmlWriter`

Работа с иерархическими данными

Рассмотрим следующие классы:

```
public class Contacts
{
    public IList<Customer> Customers = new List<Customer> ();
    public IList<Supplier> Suppliers = new List<Supplier> ();
}

public class Customer { public string FirstName, LastName; }
public class Supplier { public string Name; }
```

Предположим, что мы хотим применить классы `XmlReader` и `XmlWriter` для сериализации объекта `Contacts` в XML, как в приведенном ниже фрагменте:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<contacts>
  <customer id="1">
    <firstname>Jay</firstname>
    <lastname>Dee</lastname>
  </customer>
  <customer <!-- мы будем предполагать, что id необязателен -->
    <firstname>Kay</firstname>
    <lastname>Gee</lastname>
  </customer>
  <supplier>
    <name>X Technologies Ltd</name>
  </supplier>
</contacts>
```

Более удачный подход заключается в том, чтобы не записывать один большой метод, а инкапсулировать XML-функциональность в самих типах `Customer` и `Supplier`, реализовав для них методы `ReadXml` и `WriteXml`. Используемый шаблон довольно прост:

- когда методы `ReadXml` и `WriteXml` завершаются, они оставляют средство чтения/записи на той же глубине;
- метод `ReadXml` читает внешний элемент, тогда как метод `WriteXml` записывает только его внутреннее содержимое.

Ниже показано, как можно было бы реализовать тип `Customer`:

```
public class Customer
{
    public const string XmlName = "customer";
    public int? ID;
    public string FirstName, LastName;

    public Customer () { }
    public Customer (XmlReader r) { ReadXml (r); }

    public void ReadXml (XmlReader r)
    {
        if (r.MoveToAttribute ("id")) ID = r.ReadContentAsInt();
        r.ReadStartElement();
        FirstName = r.ReadElementContentAsString ("firstname", "");
        LastName = r.ReadElementContentAsString ("lastname", "");
        r.ReadEndElement();
    }

    public void WriteXml (XmlWriter w)
    {
        if (ID.HasValue) w.WriteAttributeString ("id", "", ID.ToString());
        w.WriteElementString ("firstname", FirstName);
        w.WriteElementString ("lastname", LastName);
    }
}
```

Обратите внимание, что метод `ReadXml` читает узлы внешнего начального и конечного элементов. Если бы эту работу делал вызывающий компонент, то класс `Customer` мог бы не читать собственные атрибуты. Причина, по которой метод `WriteXml` не сделан симметричным в этом отношении, двояка:

- вызывающий компонент может нуждаться в выборе способа именования внешнего элемента;
- вызывающему компоненту может быть необходима запись дополнительных XML-атрибутов, таких как *подтип* элемента (который затем может применяться для принятия решения о том, экземпляр какого класса создавать при чтении данного элемента).

Другое преимущество следования этому шаблону связано с тем, что ваша реализация будет совместимой с интерфейсом `IXmlSerializable` (глава 17).

Класс `Supplier` аналогичен:

```
public class Supplier
{
    public const string XmlName = "supplier";
    public string Name;
```

```

public Supplier () { }
public Supplier (XmlReader r) { ReadXml (r); }
public void ReadXml (XmlReader r)
{
    r.ReadStartElement();
    Name = r.ReadElementContentAsString ("name", "");
    r.ReadEndElement();
}
public void WriteXml (XmlWriter w)
{
    w.WriteElementString ("name", Name);
}
}

```

В классе Contacts мы должны выполнять перечисление элемента customers в методе ReadXml, проверяя, является ли каждый подэлемент заказчиком или поставщиком. Также понадобится закодировать обработку пустых элементов:

```

public void ReadXml (XmlReader r)
{
    bool isEmpty = r.IsEmptyElement; // Это обеспечивает корректную
    r.ReadStartElement(); // обработку пустого
    if (isEmpty) return; // элемента <contacts/>
    while (r.NodeType == XmlNodeType.Element)
    {
        if (r.Name == Customer.XmlName) Customers.Add (new Customer (r));
        else if (r.Name == Supplier.XmlName) Suppliers.Add (new Supplier (r));
        else
            throw new XmlException ("Unexpected node: " + r.Name);
            // Непредвиденный узел
    }
    r.ReadEndElement();
}
public void WriteXml (XmlWriter w)
{
    foreach (Customer c in Customers)
    {
        w.WriteStartElement (Customer.XmlName);
        c.WriteXml (w);
        w.WriteEndElement();
    }
    foreach (Supplier s in Suppliers)
    {
        w.WriteStartElement (Supplier.XmlName);
        s.WriteXml (w);
        w.WriteEndElement();
    }
}
}

```

Смешивание XmlReader/XmlWriter с моделью X-DOM

Переключиться на модель X-DOM можно в любой точке XML-дерева, где работа с классами XmlReader или XmlWriter становится слишком громоздкой. Использование X-DOM для обработки внутренних элементов является великолепным способом комбинирования простоты применения X-DOM и низкого расхода памяти классами XmlReader и XmlWriter.

Использование XmlReader с XElement

Чтобы прочитать текущий элемент в модель X-DOM, необходимо вызвать метод `XNode.ReadFrom`, передав ему экземпляр `XmlReader`. В отличие от `XElement.Load`, этот метод не является “жадным” в том, что он не ожидает увидеть целый документ. Взамен метод `XNode.ReadFrom` читает только до конца текущего поддерева.

В качестве примера предположим, что имеется XML-файл журнала со следующей структурой:

```
<log>
  <logentry id="1">
    <date>...</date>
    <source>...</source>
    ...
  </logentry>
  ...
</log>
```

При наличии миллиона элементов `logentry` чтение целого журнала в модель X-DOM приведет к непроизводительному расходу памяти. Более эффективное решение предусматривает обход всех элементов `logentry` с помощью класса `XmlReader` и затем использование `XElement` для индивидуальной обработки каждого элемента:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;
using (XmlReader r = XmlReader.Create ("logfile.xml", settings))
{
    r.ReadStartElement ("log");
    while (r.Name == "logentry")
    {
        XElement logEntry = (XElement) XNode.ReadFrom (r);
        int id = (int) logEntry.Attribute ("id");
        DateTime date = (DateTime) logEntry.Element ("date");
        string source = (string) logEntry.Element ("source");
        ...
    }
    r.ReadEndElement();
}
```

Если следовать шаблону, описанному в предыдущем разделе, вы можете поместить `XElement` внутрь метода `ReadXml` или `WriteXml` специального типа так, что вызывающий компонент даже не обнаружит подвоха! Например, метод `ReadXml` класса `Customer` можно было бы переписать следующим образом:

```
public void ReadXml (XmlReader r)
{
    XElement x = (XElement) XNode.ReadFrom (r);
    FirstName = (string) x.Element ("firstname");
    LastName = (string) x.Element ("lastname");
}
```

Класс `XElement` взаимодействует с классом `XmlReader`, чтобы гарантировать, что пространства имен остались незатронутыми, а префиксы соответствующим образом расширенными – даже если они определены на внешнем уровне. Таким образом, если содержимое XML-файла выглядит, как показано ниже:

```
<log xmlns="http://loggingspace">
  <logentry id="1">
    ...
```

то экземпляры `XElement`, сконструированные на уровне `logentry`, будут корректно наследовать внешнее пространство имен.

Использование `XmlWriter` с `XElement`

Класс `XElement` можно применять только для записи внутренних элементов в `XmlWriter`. В приведенном далее коде производится запись миллиона элементов `logentry` в XML-файл с использованием класса `XElement` — без помещения всех их в память:

```
using (XmlWriter w = XmlWriter.Create ("log.xml"))
{
    w.WriteStartElement ("log");
    for (int i = 0; i < 1000000; i++)
    {
        XElement e = new XElement ("logentry",
            new XAttribute ("id", i),
            new XElement ("date", DateTime.Today.AddDays (-1)),
            new XElement ("source", "test"));
        e.WriteTo (w);
    }
    w.WriteEndElement ();
}
```

С применением класса `XElement` связаны минимальные накладные расходы во время выполнения. Если мы изменим этот пример для повсеместного использования класса `XmlWriter`, то никакой заметной разницы в скорости выполнения не будет.

XSD и проверка достоверности схемы

Содержимое отдельного XML-документа почти всегда является специфичным для предметной области, как в случае документа Microsoft Word, документа с конфигурацией приложения или веб-службы. Для каждой предметной области XML-файл соответствует определенному шаблону. Для описания схем таких шаблонов предусмотрено несколько стандартов, которые предназначены для унификации и автоматизации процедур интерпретации и проверки достоверности XML-документов. Самым широко принятым стандартом является *XSD (XML Schema Definition* — определение схемы XML). Его предшественники, DTD и XDR, также поддерживаются пространством имен `System.Xml`.

Взгляните на следующий XML-документ:

```
<?xml version="1.0"?>
<customers>
  <customer id="1" status="active">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </customer>
  <customer id="1" status="archived">
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </customer>
</customers>
```

Определение XSD для этого документа можно записать так:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="customers">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="customer">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="firstname" type="xs:string" />
              <xs:element name="lastname" type="xs:string" />
            </xs:sequence>
            <xs:attribute name="id" type="xs:int" use="required" />
            <xs:attribute name="status" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Как видите, сами XSD-документы представляются с помощью XML. Более того, XSD-документ может быть описан посредством XSD — вы найдете это определение по адресу <http://www.w3.org/2001/xmlschema.xsd>.

Выполнение проверки достоверности схемы

Перед чтением или обработкой файл либо документ XML можно проверить на соответствие одной или нескольким схемам. Это делается по следующим причинам:

- можно уменьшить объем проверки на предмет ошибок и обработки исключений;
- проверка достоверности схемы позволяет обнаружить ошибки, которые в противном случае остались бы незамеченными;
- сообщения об ошибках являются подробными и информативными.

Для выполнения проверки достоверности необходимо подключить схему к объекту `XmlReader`, `XmlDocument` или `X-DOM` и затем читать либо загружать XML-данные обычным образом. Проверка достоверности посредством схемы происходит автоматически по мере чтения содержимого, так что входной поток не читается дважды.

Проверка достоверности `XmlReader`

Ниже показано, как подключить схему из файла `customers.xsd` к объекту `XmlReader`:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
...

```

Если схема является встроенной, то вместо добавления к свойству Schemas понадобится установить следующий флаг:

```
settings.ValidationFlags |= XmlSchemaValidationFlags.ProcessInlineSchema;
```

После этого можно выполнять чтение обычным образом. Если в какой-то момент происходит отказ при проверке достоверности посредством схемы, то генерируется исключение `XmlSchemaValidationException`.



Вызов метода `Read` сам по себе обеспечивает проверку достоверности и элементов, и атрибутов: переходить к каждому отдельному атрибуту с целью его проверки не придется.

Если требуется *только* проверить документ, можно поступить так:

```
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
    try { while (r.Read()) ; }
    catch (XmlSchemaValidationException ex)
    {
        ...
    }
```

Класс `XmlSchemaValidationException` имеет свойства `Message`, `LineNumber` и `LinePosition`. В этом случае он сообщает лишь о первой ошибке, обнаруженной в документе. Чтобы получить сведения обо всех ошибках в документе, потребуется организовать обработку события `ValidationEventHandler`:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");
settings.ValidationEventHandler += ValidationHandler;
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
    while (r.Read()) ;
```

Когда это событие обрабатывается, ошибки, связанные со схемой, больше не будут приводить к генерации исключения. Вместо этого они запускают обработчик события:

```
static void ValidationHandler (object sender, ValidationEventArgs e)
{
    Console.WriteLine ("Error: " + e.Exception.Message);
}
```

Свойство `Exception` класса `ValidationEventArgs` содержит экземпляр исключения `XmlSchemaValidationException`, которое сгенерировалось бы в противном случае.



В пространстве имен `System.Xml` также определен класс по имени `XmlValidatingReader`. Он предназначен для выполнения проверки достоверности схемы в версиях, предшествующих .NET Framework 2.0, и в настоящее время считается устаревшим.

Проверка достоверности X-DOM

Для выполнения проверки достоверности файла или потока XML во время его чтения в модель X-DOM необходимо создать экземпляр `XmlReader`, подключить схемы и применить средство чтения для загрузки DOM-модели:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");

XDocument doc;
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
    try { doc = XDocument.Load (r); }
    catch (XmlSchemaValidationException ex) { ... }
```

С помощью расширяющих методов из пространства имен `System.Xml.Schema` можно выполнять проверку достоверности объекта `XDocument` или `XElement`, уже находящегося в памяти. Эти методы принимают экземпляр `XmlSchemaSet` (коллекция схем) и обработчик событий проверки:

```
XDocument doc = XDocument.Load (@"customers.xml");
XmlSchemaSet set = new XmlSchemaSet ();
set.Add (null, @"customers.xsd");
StringBuilder errors = new StringBuilder ();
doc.Validate (set, (sender, args) => { errors.AppendLine
                                     (args.Exception.Message); }
             );
Console.WriteLine (errors.ToString());
```

XSLT

Аббревиатура XSLT означает *Extensible Stylesheet Language Transformations* (расширяемый язык трансформации таблиц стилей). XSLT представляет собой язык XML, который описывает преобразование одного XML-текста в другой. Наиболее типичным примером такого преобразования служит трансформация XML-документа (который обычно описывает данные) в XHTML-документ (описывающий форматированный документ).

Рассмотрим следующий XML-файл:

```
<customer>
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Показанный ниже XSLT-файл описывает такое преобразование:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <html>
      <p><xsl:value-of select="//firstname"/></p>
      <p><xsl:value-of select="//lastname"/></p>
    </html>
  </xsl:template>
</xsl:stylesheet>
```


Вывод выглядит так:

```
<html>
  <p>Jim</p>
  <p>Bo</p>
</html>
```

Класс `System.Xml.Xsl.XslCompiledTransform` эффективно выполняет XSLT-преобразования. Он является заменой устаревшему классу `XmlTransform`. Класс `XmlTransform` работает очень просто:

```
XslCompiledTransform transform = new XslCompiledTransform();
transform.Load ("test.xslt");
transform.Transform ("input.xml", "output.xml");
```

Обычно удобнее пользоваться перегруженной версией метода `Transform`, которая вместо выходного файла принимает объект `XmlWriter`, что позволяет управлять форматированием.