ГЛАВА 24

Введение в LINQ to XML

Как разработчик приложений .NET, вы будете встречать данные, основанные на XML, в самых разнообразных местах. Конфигурационные файлы для обычных и веб-приложений хранят информацию в формате XML. Инфраструктура Windows Presentation Foundation использует основанную на XML грамматику (XAML) для представления настольных графических пользовательских интерфейсов. Объекты `DataSet` из ADO.NET могут легко сохранять (или загружать) данные в формате XML. Даже инфраструктура Windows Communication Foundation хранит многочисленные параметры в виде корректно сформатированных строк XML.

Хотя данные XML действительно вездесущи, исторически сложилось так, что программирование с применением XML было утомительным, громоздким и очень сложным, если не владеть довольно большим числом технологий XML (XPath, XQuery, XSLT, DOM, SAX и т.д.). В самом первом выпуске .NET была предоставлена специальная сборка по имени `System.Xml.dll`, предназначенная для программирования с использованием документов XML. В ней находится несколько пространств имен и типов для разнообразных технологий программирования XML, а также ряд API-интерфейсов XML, специфичных для .NET, таких как классы `XmlReader/XmlWriter`.

В наши дни большинство программистов предпочитают взаимодействовать с данными XML с применением API-интерфейса LINQ to XML. Вы увидите в этой главе, что программная модель LINQ to XML позволяет выражать структуру данных XML в коде и предлагает намного более простой способ создания, манипулирования, загрузки и сохранения данных XML. Наряду с тем, что LINQ to XML можно использовать только в качестве упрощенного способа создания документов XML, для быстрого запрашивания информации из них довольно легко задействовать выражения запросов LINQ.

История о двух API-интерфейсах XML

Когда появилась первая версия платформы .NET, программисты имели возможность манипулировать документами XML с применением типов из сборки `System.Xml.dll`. Используя содержащиеся в ней пространства имен и типы, можно было генерировать данные XML в памяти и сохранять их в дисковом хранилище. Вдобавок сборка `System.Xml.dll` предоставляла типы, позволяющие загружать документ XML в память, искать в нем специфические узлы, проверять документ на соответствие заданной схеме и выполнять другие распространенные задачи.

Несмотря на то что эта первоначальная библиотека успешно применялась во многих проектах .NET, работа с ее типами была несколько запутанной (мягко говоря), поскольку программная модель не имела никакого отношения к структуре самого документа XML. Например, пусть необходимо создать документ XML в памяти и сохранить его в файловой системе. В случае использования типов из `System.Xml.dll` можно написать код, подобный показанному ниже (первым делом нужно создать новый проект консоль-

ного приложения по имени LinqToXmlFirstLook и импортировать пространство имен System.Xml):

```
private static void BuildXmlDocWithDOM()
{
  // Создать новый документ XML в памяти.
  XmlDocument doc = new XmlDocument();

  // Заполнить документ корневым элементом по имени <Inventory>.
  XmlElement inventory = doc.CreateElement("Inventory");

  // Создать подэлемент по имени <Car> с атрибутом ID.
  XmlElement car = doc.CreateElement("Car");
  car.SetAttribute("ID", "1000");

  // Построить данные внутри элемента <Car>.
  XmlElement name = doc.CreateElement("PetName");
  name.InnerText = "Jimbo";
  XmlElement color = doc.CreateElement("Color");
  color.InnerText = "Red";
  XmlElement make = doc.CreateElement("Make");
  make.InnerText = "Ford";

  // Добавить к элементу <Car> элементы <PetName>, <Color> и <Make>.
  car.AppendChild(name);
  car.AppendChild(color);
  car.AppendChild(make);

  // Добавить к элементу <Inventory> элемент <Car>.
  inventory.AppendChild(car);

  // Вставить завершенный XML в объект XmlDocument и сохранить в файле.
  doc.AppendChild(inventory);
  doc.Save("Inventory.xml");
}
```

Вызвав этот метод внутри Main(), можно увидеть, что файл Inventory.xml (находящийся в папке bin\Debug) содержит следующие данные:

```
<Inventory>
 <Car ID="1000">
  <PetName>Jimbo</PetName>
  <Color>Red</Color>
  <Make>Ford</Make>
 </Car>
</Inventory>
```

Хотя метод BuildXmlDocWithDOM() работает ожидаемым образом, уместно сделать несколько замечаний. Программная модель System.Xml.dll — это реализация от Microsoft спецификации W3C DOM (Document Object Model — объектная модель документа). Согласно такой модели документ XML строится снизу вверх. Сначала создается документ, затем элементы и, наконец, элементы добавляются в документ. Чтобы выразить это в коде, потребуется написать довольно много вызовов методов из классов XmlDocument и XmlElement (помимо прочих).

В приведенном примере для построения очень простого документа XML понадобилось 16 строк кода (без учета комментариев). Создание более сложного документа с помощью сборки System.Xml.dll требует написания гораздо большего объема кода. Несмотря на то что код определенно можно упростить, выполняя построение узлов посредством циклических и условных конструкций, факт остается фактом — тело кода имеет минимум визуального отражения финального дерева XML.

Интерфейс LINQ to XML как лучшая модель DOM

Альтернативный способ построения, манипулирования и запрашивания документов XML предлагает API-интерфейс LINQ to XML, в котором применяется намного более функциональный подход по сравнению с моделью DOM из пространства имен `System.Xml`. Вместо построения документа XML за счет индивидуального формирования элементов и обновления дерева XML через набор вызовов методов код пишется сверху вниз:

```
private static void BuildXmlDocWithLINQToXml()
{
  // Создать документ XML в более "функциональной" манере.
  XElement doc =
   new XElement("Inventory",
    new XElement("Car", new XAttribute("ID", "1000"),
      new XElement("PetName", "Jimbo"),
      new XElement("Color", "Red"),
      new XElement("Make", "Ford")
    )
  );
  // Сохранить документ в файле.
  doc.Save("InventoryWithLINQ.xml");
}
```

Здесь используется новый набор типов из пространства имен `System.Xml.Linq`, а именно — `XElement` и `XAttribute`. Вызов метода `BuildXmlDocWithLINQToXml()` внутри `Main()` приводит к получению тех же самых данных XML, но на этот раз с гораздо меньшими усилиями. Обратите внимание, что благодаря аккуратным отступам исходный код теперь имеет ту же общую структуру, что и результирующий документ XML. Это очень удобно и само по себе, но вдобавок оцените, насколько компактнее данный код по сравнению с предыдущим примером (сэкономлено около 10 строк).

В коде не применяются выражения запросов LINQ, а просто с помощью типов из пространства имен `System.Xml.Linq` в памяти генерируется документ XML, который затем сохраняется в файле. Фактически API-интерфейс LINQ to XML использовался как лучшая модель DOM. Позже в главе вы увидите, что классы из `System.Xml.Linq` поддерживают LINQ и могут быть целью для той же разновидности запросов LINQ, которая рассматривалась в главе 12.

По мере изучения LINQ to XML, скорее всего, вы начнете отдавать предпочтение этому API-интерфейсу перед первоначальными библиотеками XML платформы .NET. Это не означает, что вы никогда не станете применять пространство имен из библиотеки `System.Xml.dll`, но случаев выбора `System.Xml.dll` для новых проектов будет значительно меньше.

Синтаксис литералов Visual Basic как лучший способ работы с LINQ to XML

Прежде чем приступить к формальным исследованиям LINQ to XML с точки зрения языка C#, имеет смысл кратко упомянуть о том, что язык Visual Basic переносит функциональный подход этого API-интерфейса на следующий уровень. В Visual Basic можно определять *литералы XML*, которые позволяют присваивать объекту `XElement` поток встроенной разметки XML прямо в коде. Предполагая, что есть проект VB, можно создать следующий метод:

```
Public Class XmlLiteralExample
  Public Sub MakeXmlFileUsingLiterals()
    ' Обратите внимание на возможность встраивания данных XML в XElement.
    Dim doc As XElement =
      <Inventory>
        <Car ID="1000">
          <PetName>Jimbo</PetName>
          <Color>Red</Color>
          <Make>Ford</Make>
        </Car>
      </Inventory>
    ' Сохранить в файле.
    doc.Save("InventoryVBStyle.xml")
  End Sub
End Class
```

После того, как компилятор VB обработал литерал XML, он отобразит данные XML на корректную внутреннюю объектную модель LINQ to XML. На самом деле во время работы с LINQ to XML внутри проекта VB синтаксис литералов XML воспринимается IDE-средой просто как сокращенное обозначение связанного кода. На рис. 24.1 обратите внимание, что применение операции точки к закрывающему дескриптору `</Inventory>` приводит к отображению тех же самых членов, как и в случае применения этой операции к строго типизированному `XElement`.

Хотя эта книга посвящена языку программирования C#, некоторые разработчики считают, что поддержка XML в VB является непревзойденной. Даже если вы из тех, кто не может представить себе работу с языком из семейства BASIC, все равно рекомендуется изучить синтаксис литералов VB в документации .NET Framework 4.6 SDK. Все процедуры манипуляций с данными XML могут быть вынесены в отдельную сборку `*.dll`, так что вполне допустимо использовать для них VB!

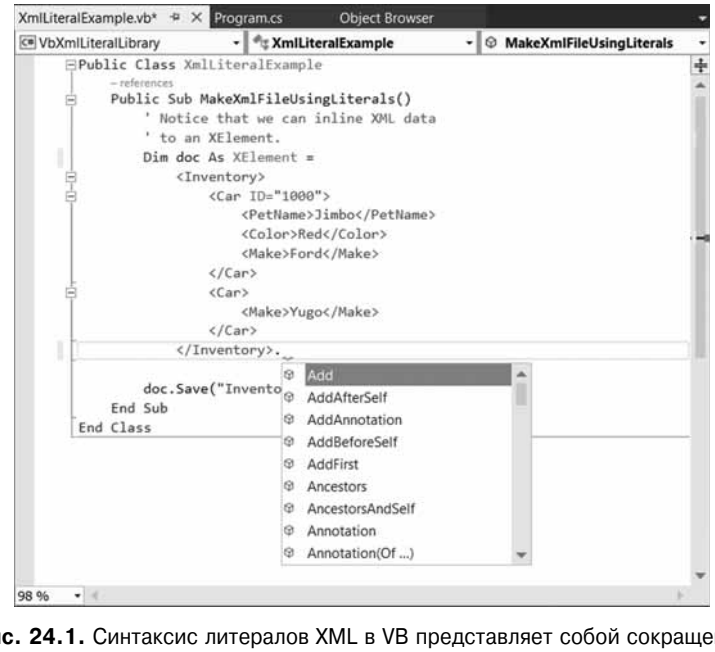

Рис. 24.1. Синтаксис литералов XML в VB представляет собой сокращение для работы с объектной моделью LINQ to XML

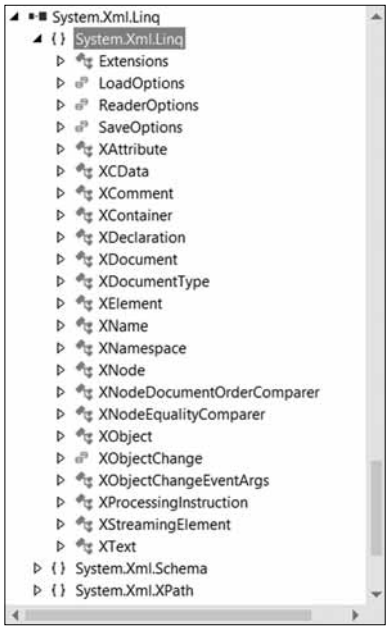

Рис. 24.2. Пространства имен `System.Xml.Linq.dll`

Члены пространства имен `System.Xml.Linq`

Несколько неожиданно, но в основной сборке LINQ to XML (`System.Xml.Linq.dll`) определено относительно небольшое количество типов в трех разных пространствах имен: `System.Xml.Linq`, `System.Xml.Schema` и `System.Xml.XPath` (рис. 24.2).

Главное пространство имен, `System.Xml.Linq`, содержит управляемый набор классов, которые представляют разнообразные аспекты документа XML (элементы и их атрибуты, пространства имен XML, комментарии XML, инструкции обработки и т.д.). В табл. 24.1 кратко описаны избранные основные члены `System.Xml.Linq`.

На рис. 24.3 показана цепочка наследования основных классов.

Таблица 24.1. Избранные члены пространства имен `System.Xml.Linq`

Член	Описание
`XAttribute`	Представляет атрибут XML заданного элемента XML
`XCData`	Представляет раздел `CDATA` в документе XML. Информация в разделе `CDATA` — это данные документа XML, которые должны быть включены, но не отвечают правилам грамматики XML (например, код сценария)
`XComment`	Представляет комментарий XML
`XDeclaration`	Представляет открывающее объявление документа XML
`XDocument`	Представляет полный документ XML
`XElement`	Представляет заданный элемент внутри документа XML, включая корневой
`XName`	Представляет имя элемента или атрибута XML
`XNamespace`	Представляет пространство имен XML
`XNode`	Представляет абстрактную концепцию узла (элемент, комментарий, тип документа, инструкция обработки или текстовый узел) в дереве XML
`XProcessingInstruction`	Представляет инструкцию обработки XML
`XStreamingElement`	Представляет элементы в дереве XML, которые поддерживают отложенный потоковый вывод

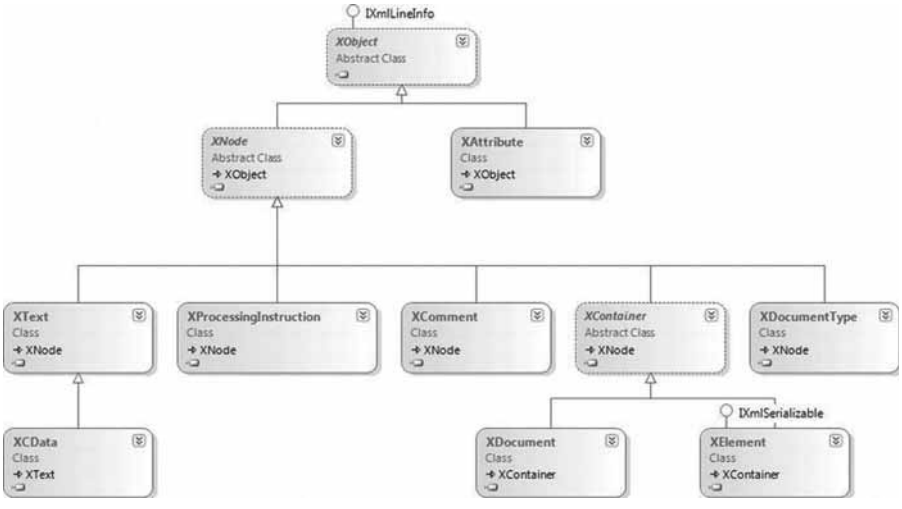

Рис. 24.3. Иерархия основных классов LINQ to XML

Осевые методы LINQ to XML

В дополнение к классам `X*` внутри пространства имен `System.Xml.Linq` определен класс по имени `Extensions`, определяющий набор расширяющих методов, которые обычно расширяют `IEnumerable<T>`, где `T` — потомок `XNode` или `XContainer`. В табл. 24.2 описаны важные расширяющие методы, о которых следует знать (как вы увидите, они удобны при работе с запросами LINQ).

Таблица 24.2. Избранные члены класса LINQ to XML

Член	Описание
`Ancestor<T>()`	Возвращает отфильтрованную коллекцию элементов, которая содержит предков каждого узла в исходной коллекции
`Attributes()`	Возвращает отфильтрованную коллекцию атрибутов каждого элемента в исходной коллекции
`DescendantNodes<T>`	Возвращает коллекцию узлов-потомков каждого документа и элемента в исходной коллекции
`Descendants<T>`	Возвращает отфильтрованную коллекцию элементов, которая содержит элементы-потомки каждого элемента и документа в исходной коллекции
`Elements<T>`	Возвращает коллекцию дочерних элементов каждого элемента и документа в исходной коллекции
`Nodes<T>`	Возвращает коллекцию дочерних узлов каждого документа и элемента в исходной коллекции
`Remove()`	Удаляет каждый атрибут исходной коллекции из родительского элемента
`Remove<T>()`	Удаляет все вхождения заданного узла в исходной коллекции

Как можно понять по именам, эти методы позволяют запрашивать загруженное дерево XML в поисках элементов, атрибутов и их значений. Все вместе такие методы называются *осевыми методами* или просто *осями*. Их можно применять напрямую к частям дерева узлов либо использовать для построения более сложных запросов LINQ.

На заметку! Абстрактный класс `XContainer` поддерживает несколько методов, которые именованы идентично членам класса `Extensions`. Класс `XContainer` является родительским для `XElement` и `XDocument`, поэтому оба класса поддерживают ту же самую общую функциональность.

Примеры применения некоторых осевых методов будут встречаться далее в главе. Пока что вот краткий пример:

```
private static void DeleteNodeFromDoc()
{
  XElement doc =
    new XElement("Inventory",
      new XElement("Car", new XAttribute("ID", "1000"),
        new XElement("PetName", "Jimbo"),
        new XElement("Color", "Red"),
        new XElement("Make", "Ford")
      )
    );
  // Удалить элемент PetName из дерева.
  doc.Descendants("PetName").Remove();
    Console.WriteLine(doc);
}
```

В результате вызова этого метода получается следующее "усеченное" дерево XML:

```
<Inventory>
 <Car ID="1000">
  <Color>Red</Color>
  <Make>Ford</Make>
 </Car>
</Inventory>
```

Странность класса XName (и XNamespace)

Просмотрев сигнатуры осевых методов LINQ to XML (или идентично именованных членов `XContainer`), вы заметите, что они обычно требуют указания того, что выглядит как объект `XName`. Взгляните на сигнатуру метода `Descendants()`, определенного в `XContainer`:

```
public IEnumerable<XElement> Descendants(XName name)
```

Класс `XName` является "странным" в том, что он никогда не будет использоваться в коде напрямую. В действительности, поскольку этот класс не имеет открытого конструктора, объект типа `XName` создавать невозможно:

```
// Ошибка! Объекты XName создавать невозможно!
doc.Descendants(new XName("PetName")).Remove();
```

В формальном определении `XName` вы обнаружите специальную операцию неявного преобразования (специальные операции преобразования были описаны в главе 11), которая отобразит простой тип `System.String` на корректный объект `XName`:

```
// В действительности объект XName создается на заднем плане!
doc.Descendants("PetName").Remove();
```

На заметку! Класс `XNamespace` также поддерживает ту же самую разновидность неявного преобразования строк.

Хорошая новость в том, что при работе с осевыми методами можно применять текстовые значения для представления имен элементов или атрибутов и позволять API-интерфейсу LINQ to XML отображать строковые данные на необходимые типы объектов.

Исходный код. Проект `LinqToXmlFirstLook` доступен в подкаталоге `Chapter_24`.

Работа с `XElement` и `XDocument`

Давайте продолжим исследование LINQ to XML, создав новый проект консольного приложения по имени `ConstructingXmlDocs`. После создания проекта импортируем пространство имен `System.Xml.Linq` в начальный файл кода. Вы уже видели, что класс `XDocument` представляет полный документ XML в программной модели LINQ to XML, т.к. он может использоваться для определения корневого элемента, а также всех содержащихся в нем элементов, инструкций обработки и объявлений XML. Вот еще один пример построения данных XML с применением `XDocument`:

```
static void CreateFullXDocument()
{
  XDocument inventoryDoc =
    new XDocument(
      new XDeclaration("1.0", "utf-8", "yes"),
      new XComment("Current Inventory of cars!"),
      new XProcessingInstruction("xml-stylesheet",
        "href='MyStyles.css' title='Compact' type='text/css'"),
      new XElement("Inventory",
        new XElement("Car", new XAttribute("ID", "1"),
          new XElement("Color", "Green"),
          new XElement("Make", "BMW"),
          new XElement("PetName", "Stan")
        ),
        new XElement("Car", new XAttribute("ID", "2"),
          new XElement("Color", "Pink"),
          new XElement("Make", "Yugo"),
          new XElement("PetName", "Melvin")
        )
      )
    );
  // Сохранить на диске.
  inventoryDoc.Save("SimpleInventory.xml");
}
```

И снова обратите внимание, что конструктор объекта `XDocument` на самом деле представляет собой дерево дополнительных объектов LINQ to XML. Вызываемый здесь конструктор принимает в первом параметре объект `XDeclaration`, за которым следует параметр-массив объектов (вспомните, что параметры-массивы позволяют передавать разделенные запятыми списки аргументов, которые автоматически упаковываются в массив):

```
public XDocument(System.Xml.Linq.XDeclaration declaration,
                 params object[] content)
```

В результате вызова этого метода внутри `Main()` в файл `SimpleInventory.xml` будут записаны приведенные ниже данные:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--Current Inventory of cars!-->
<?xml-stylesheet href='MyStyles.css' title='Compact' type='text/css'?>
<Inventory>
 <Car ID="1">
  <Color>Green</Color>
  <Make>BMW</Make>
  <PetName>Stan</PetName>
 </Car>
 <Car ID="2">
  <Color>Pink</Color>
  <Make>Yugo</Make>
  <PetName>Melvin</PetName>
 </Car>
</Inventory>
```

Как выясняется, стандартное объявление XML для любого `XDocument` предусматривает использование кодировки UTF-8, XML версии 1.0 и автономного документа. Следовательно, код создания объекта `XDeclaration` можно полностью удалить и получить те же самые данные; учитывая, что почти любой документ требует одного и того же объявления, применять `XDeclaration` приходится нечасто.

Если определять инструкции обработки или специальное объявление XML нет необходимости, то можно вообще избежать использования `XDocument` и просто работать с классом `XElement`. Помните, что `XElement` может применяться для представления корневого элемента документа XML и всех подобъектов. Таким образом, вот как можно сгенерировать прокомментированный список складских запасов:

```
static void CreateRootAndChildren()
{
  XElement inventoryDoc =
    new XElement("Inventory",
      new XComment("Current Inventory of cars!"),
        new XElement("Car", new XAttribute("ID", "1"),
          new XElement("Color", "Green"),
          new XElement("Make", "BMW"),
          new XElement("PetName", "Stan")
        ),
        new XElement("Car", new XAttribute("ID", "2"),
          new XElement("Color", "Pink"),
          new XElement("Make", "Yugo"),
          new XElement("PetName", "Melvin")
        )
      )
    );
  // Сохранить на диске.
  inventoryDoc.Save("SimpleInventory.xml");
}
```

Результат будет более или менее идентичным кроме инструкции обработки для гипотетической стилевой таблицы:

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory>
 <!--Current Inventory of cars!-->
 <Car ID="1">
  <Color>Green</Color>
```

```
  <Make>BMW</Make>
  <PetName>Stan</PetName>
 </Car>
 <Car ID="2">
  <Color>Pink</Color>
  <Make>Yugo</Make>
  <PetName>Melvin</PetName>
</Car>
</Inventory>
```

Генерация документов из массивов и контейнеров

До сих пор документы XML строились с использованием жестко закодированных значений для конструктора. Но гораздо чаще требуется генерировать объект `XElement` (или `XDocument`), читая данные из массивов, объектов ADO.NET, файлов данных и т.п. Один из способов отображения данных из памяти на новый объект `XElement` заключается в применении стандартных циклов `for` для перемещения данных в объектную модель LINQ to XML. Хотя это определенно возможно, проще встроить запрос LINQ прямо в код конструирования `XElement`.

Предположим, что имеется анонимный массив анонимных классов (просто чтобы сократить объем кода в примере; подойдет также любой массив, `List<T>` или другой контейнер). Отобразить эти данные на `XElement` можно следующим образом:

```
static void MakeXElementFromArray()
{
  // Создать анонимный массив анонимных типов.
  var people = new[] {
    new { FirstName = "Mandy", Age = 32},
    new { FirstName = "Andrew", Age = 40 },
    new { FirstName = "Dave", Age = 41 },
    new { FirstName = "Sara", Age = 31}
  };
XElement peopleDoc =
  new XElement("People",
    from c in people select new XElement("Person", new XAttribute("Age",
c.Age),
            new XElement("FirstName", c.FirstName))
  );
  Console.WriteLine(peopleDoc);
}
```

Здесь объект `peopleDoc` определяет корневой элемент `<People>` с результатами запроса LINQ. Этот запрос LINQ создает новые объекты `XElement` на основе каждого элемента в массиве `people`. Если встроенный запрос покажется трудным для восприятия, все можно разделить на явные шаги:

```
static void MakeXElementFromArray()
{
  // Создать анонимный массив анонимных типов.
  var people = new[] {
    new { FirstName = "Mandy", Age = 32},
    new { FirstName = "Andrew", Age = 40 },
    new { FirstName = "Dave", Age = 41 },
    new { FirstName = "Sara", Age = 31}
  };
```

```
  var arrayDataAsXElements = from c in people
    select
      new XElement("Person",
        new XAttribute("Age", c.Age),
        new XElement("FirstName", c.FirstName));
  XElement peopleDoc = new XElement("People", arrayDataAsXElements);
  Console.WriteLine(peopleDoc);
}
```

В любом случае вывод будет таким:

```
<People>
 <Person Age="32">
  <FirstName>Mandy</FirstName>
 </Person>
 <Person Age="40">
  <FirstName>Andrew</FirstName>
 </Person>
 <Person Age="41">
  <FirstName>Dave</FirstName>
 </Person>
 <Person Age="31">
  <FirstName>Sara</FirstName>
 </Person>
</People>
```

Загрузка и разбор содержимого XML

Типы `XElement` и `XDocument` поддерживают методы `Load()` и `Parse()`, которые позволяют наполнить объектную модель XML из объектов `string`, содержащих данные XML, либо из внешних файлов XML. Рассмотрим показанный далее метод, иллюстрирующий оба подхода:

```
static void ParseAndLoadExistingXml()
{
  // Построить объект XElement из строки.
  string myElement =
    @"<Car ID ='3'>
       <Color>Yellow</Color>
       <Make>Yugo</Make>
      </Car>";
  XElement newElement = XElement.Parse(myElement);
  Console.WriteLine(newElement);
  Console.WriteLine();

  // Загрузить файл SimpleInventory.xml.
  XDocument myDoc = XDocument.Load("SimpleInventory.xml");
  Console.WriteLine(myDoc);
}
```

Исходный код. Проект `ConstructingXmlDocs` доступен в подкаталоге `Chapter_24`.

Манипулирование документом XML в памяти

Итак, к настоящему моменту вы видели разнообразные способы использования LINQ to XML для создания, сохранения, разбора и загрузки данных XML. Следующий аспект LINQ to XML, который необходимо исследовать — навигация по документу с целью поиска местоположения и изменения специфичных элементов дерева с применением запросов LINQ и осевых методов LINQ to XML.

Для этого мы построим приложение Windows Forms, которое будет отображать данные из документа XML, сохраненного на жестком диске. Графический пользовательский интерфейс позволит пользователю вводить данные для нового узла, который будет добавлен к тому же самому документу XML. Наконец, пользователю будет предоставлено несколько способов для выполнения поиска в документе с помощью набора запросов LINQ.

На заметку! Учитывая, что некоторые запросы LINQ уже строились в главе 12, здесь они повторяться не будут. В разделе "Querying XML Trees" ("Запрашивание деревьев XML") документации .NET Framework 4.6 SDK можно ознакомиться с дополнительными примерами LINQ to XML.

Построение пользовательского интерфейса для приложения LINQ to XML

Создадим проект приложения Windows Forms под названием `LinqToXmlWinApp` и изменим имя первоначального файла `Form1.cs` на `MainForm.cs` (в окне Solution Explorer). Графический пользовательский интерфейс этого приложения довольно прост. В левой части окна находится элемент управления `TextBox` (по имени `txtInventory`), свойство `Multiline` которого установлено в `true`, а свойство `ScrollBars` — в `Both`.

Кроме того, имеется одна группа простых элементов управления `TextBox` (`txtMake`, `txtColor` и `txtPetName`) и элемент `Button` (`btnAddNewItem`), щелчок на котором приводит к добавлению новой записи в документ XML. Наконец, есть еще одна группа элементов управления (`TextBox` по имени `txtMakeToLookUp` и элемент `Button` с именем `btnLookUpColors`), которая позволяет запрашивать из документа XML набор указанных узлов. На рис. 24.4 показана возможная компоновка окна.

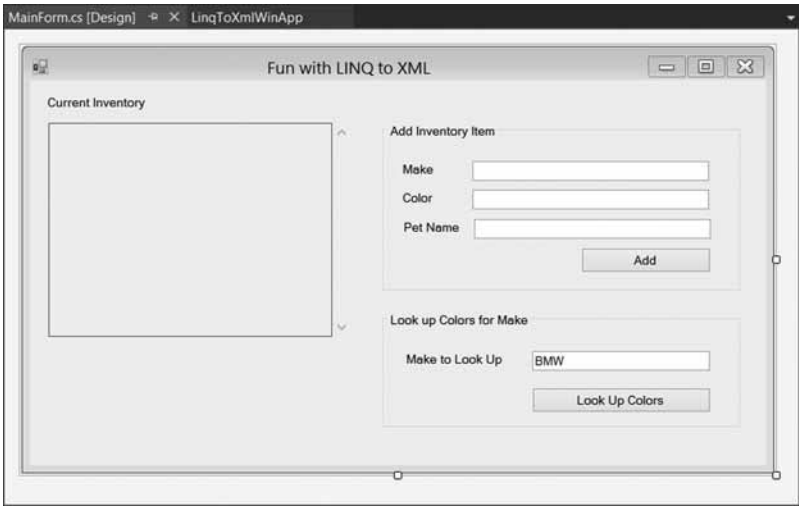

Рис. 24.4. Графический пользовательский интерфейс для приложения LINQ to XML

Потребуется обработать событие Click каждой кнопки для генерации методов обработки событий, а также обработать событие Load формы. Мы займемся этим чуть позже.

Импортирование файла Inventory.xml

В состав загружаемого кода примеров для книги включен файл Inventory.xml, в котором имеется набор сущностей внутри корневого элемента <Inventory>. Импортируем этот файл в проект, выбрав пункт меню Project⇨Add Existing Item (Проект⇨Добавить существующий элемент). Просмотрев данные, можно заметить, что корневой элемент определяет набор элементов <Car>, каждый из которых определен подобно следующему:

```
<Car carID ="0">
 <Make>Ford</Make>
 <Color>Blue</Color>
 <PetName>Chuck</PetName>
</Car>
```

Прежде чем продолжить, этот файл понадобится выбрать в окне Solution Explorer и затем в окне Properties (Свойства) установить свойство Copy to Output Directory (Копировать в выходной каталог) в Copy Always (Копировать всегда). Это обеспечит помещение данных в папку bin\Debug при компиляции приложения.

Определение вспомогательного класса LINQ to XML

Чтобы изолировать данные LINQ to XML, добавим в проект новый класс по имени LinqToXmlObjectModel. В нем будет определен набор статических методов, инкапсулирующих некоторую логику LINQ to XML. Первым делом определим метод, который возвращает заполненный объект XDocument на основе содержимого файла Inventory.xml (в новый файл должны быть импортированы пространства имен System.Xml.Linq и System.Windows.Forms):

```
public static XDocument GetXmlInventory()
{
  try
  {
    XDocument inventoryDoc = XDocument.Load("Inventory.xml");
    return inventoryDoc;
  }
  catch (System.IO.FileNotFoundException ex)
  {
    MessageBox.Show(ex.Message);
    return null;
  }
}
```

Метод InsertNewElement(), код которого приведен ниже, получает значения элементов управления TextBox из раздела Add Inventory Item (Добавить элемент на склад) и помещает новый узел внутрь элемента <Inventory>, используя осевой метод Descandants(). После этого документ будет сохранен.

```
public static void InsertNewElement(string make, string color, string
petName)
{
  // Загрузить текущий документ.
  XDocument inventoryDoc = XDocument.Load("Inventory.xml");
```

```
  // Сгенерировать случайное число для идентификатора.
  Random r = new Random();
  // Создать новый объект XElement на основе входных параметров.
  XElement newElement = new XElement("Car", new XAttribute("ID", r.Next(50000)),
    new XElement("Color", color),
    new XElement("Make", make),
    new XElement("PetName", petName));
  // Добавить к объекту XDocument в памяти.
  inventoryDoc.Descendants("Inventory").First().Add(newElement);
  // Сохранить изменения на диске.
  inventoryDoc.Save("Inventory.xml");
}
```

Финальный метод, `LookUpColorsForMake()`, получает данные из последнего элемента управления `TextBox` и с применением запроса LINQ строит строку, которая содержит цвета указанного изготовителя. Взгляните на следующую реализацию:

```
public static void LookUpColorsForMake(string make)
{
  // Загрузить текущий документ.
  XDocument inventoryDoc = XDocument.Load("Inventory.xml");
  // Найти цвета для заданного изготовителя.
  var makeInfo = from car in inventoryDoc.Descendants("Car")
    where (string)car.Element("Make") == make
    select car.Element("Color").Value;
  // Построить строку, представляющую каждый цвет.
  string data = string.Empty;
  foreach (var item in makeInfo.Distinct())
  {
    data += string.Format("- {0}\n", item);
  }
  // Показать цвета.
  MessageBox.Show(data, string.Format("{0} colors:", make));
}
```

Связывание пользовательского интерфейса и вспомогательного класса

К настоящему моменту все, что осталось сделать — это реализовать обработчики событий. Задача сводится к простым вызовам статических вспомогательных методов:

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }
  private void MainForm_Load(object sender, EventArgs e)
  {
    // Отобразить текущий документ XML склада в элементе управления TextBox.
    txtInventory.Text = LinqToXmlObjectModel.GetXmlInventory().ToString();
  }
  private void btnAddNewItem_Click(object sender, EventArgs e)
  {
    // Добавить к документу новый элемент.
    LinqToXmlObjectModel.InsertNewElement(txtMake.Text,
                                          txtColor.Text, txtPetName.Text);
```

```
    // Отобразить текущий документ XML для склада в элементе управления TextBox.
    txtInventory.Text = LinqToXmlObjectModel.GetXmlInventory().ToString();
  }
  private void btnLookUpColors_Click(object sender, EventArgs e)
  {
    LinqToXmlObjectModel.LookUpColorsForMake(txtMakeToLookUp.Text);
  }
}
```

На рис. 24.5 показан результат добавления новой записи об автомобиле и поиска всех записей для BMW.

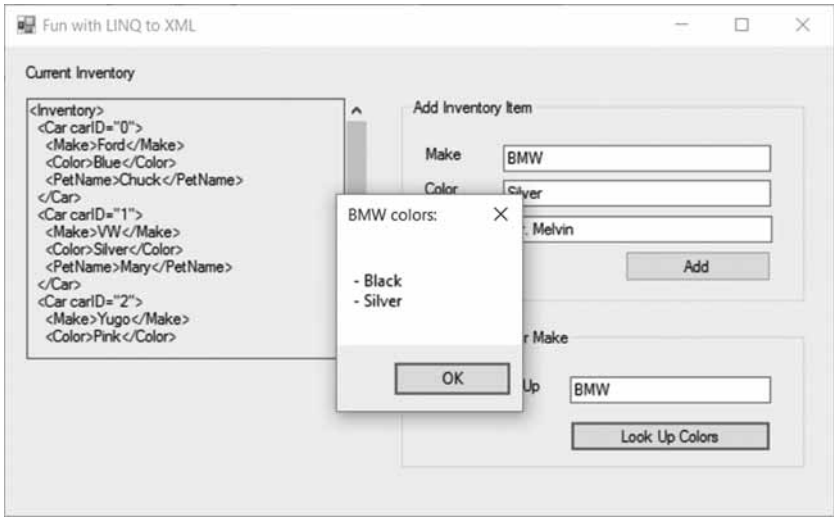

Рис. 24.5. Готовое приложение LINQ to XML

На этом завершается начальное знакомство с LINQ to XML, равно как и исследование LINQ. Впервые вы столкнулись с технологией LINQ в главе 12, где изучали LINQ to Objects. В главе 19 были представлены различные примеры использования PLINQ, а в главе 23 рассматривалось применение LINQ к сущностным объектам ADO.NET. Теперь вы готовы, да и должны двигаться дальше. В Microsoft совершенно ясно дают понять, что по мере расширения платформы .NET технология LINQ продолжит развиваться.

Исходный код. Проект `LinqToXmlWinApp` доступен в подкаталоге `Chapter_24`.

Резюме

В этой главе рассматривалась роль LINQ to XML. Вы увидели, что этот API-интерфейс является альтернативой первоначальной библиотеке для манипуляций данными XML, `System.Xml.dll`, которая поставлялась в составе платформы .NET. С помощью `System.Xml.Linq.dll` появляется возможность генерации новых документов XML с использованием подхода “сверху вниз”, при котором структура кода очень напоминает финальные данные XML. В таком свете LINQ to XML — лучшая модель DOM. Вдобавок вы узнали, каким образом строить объекты `XDocument` и `XElement` разнообразными способами (разбор, загрузка из файла, отображение объектов в памяти), а также выполнять навигацию и манипулировать данными с применением запросов LINQ.