ГЛАВА 3

Манипулирование строками

В предыдущей главе вы изучили основы хранения данных и работы с ними в среде .NET, включая различие между типами для значений и ссылочными типами. Среда .NET имеет три главных типа данных: числовые, специальные и строковые. Предыдущая глава посвящалась числовым типам. В этой главе основное внимание уделяется типу `string`.

Как вы узнаете в этой главе, тип `string` имеет некоторые специальные характеристики. Если бы вы рассмотрели биты и байты типа `string`, то, вероятно, не поняли бы, что они представляют символы. В абстрактном описании тип `string` — это тип числа со специальной грамматикой. Так как компьютер понимает только числа, он использует таблицы подстановок, которые соотносят набор символов с набором чисел.

Пример этой главы — многоязычная программа перевода. Программа эта несложная, с посредственными способностями. Но она проиллюстрирует те проблемы, с которыми вы можете столкнуться при работе со строками.

Еще один важный момент: проблемы нескольких языков становятся все более и более важными для разработчиков программного обеспечения. Современные программы редко существуют только на одном языке, поскольку рынок диктует необходимость получения прибыли с международной аудитории. Вы не обязаны знать много языков, чтобы стать успешным программистом. Но при разработке приложений, которые переводятся на разные языки, необходимо учитывать потенциальные различия между ними.

Организация приложения перевода

Как было подчеркнуто в предыдущей главе, первый шаг в разработке приложения подразумевает организацию. Теперь мы должны понять и определить средства типичного приложения, которое мы собираемся разрабатывать. Программа многоязычного перевода реализует следующие возможности.

- Переводить приветствия на три разных языка: французский, немецкий и английский.
- Переводить числа с каждого из трех языков на остальные языки.
- Переводить даты с каждого из трех языков на остальные языки.

Первая возможность, вероятно, покажется вам вполне логичной, но вторая и третья — не вполне очевидны. Под переводом мы обычно понимаем преобразование слов или фраз одного языка на другой. Но числа и даты на разных языках также могут представляться по-разному. Перевод будет означать две вещи — перевод слов с одного языка на другой и перевод чисел и дат с одного языка на другой.

Как мы уже делали в главе 2 “Типы и значения чисел .NET”, необходимо создать решение и три его части (или проекта): приложение Windows, консольное приложение проверки и библиотеку классов. После того как вы создадите все проекты, ваше рабочее пространство должно выглядеть как на рис. 3.1. Не забудьте добавить ссылки на библиотеку классов `LanguageTranslator` (щелкните правой кнопкой мыши на пункте References (Ссылки) и в появившемся контекстном меню выберите пункты Add Reference⇨Project⇨LanguageTranslator (Добавить ссылку⇨Проект⇨`LanguageTranslator`)). Не забудьте также устанавливать проект `TestLanguageTranslator` как стартовый.

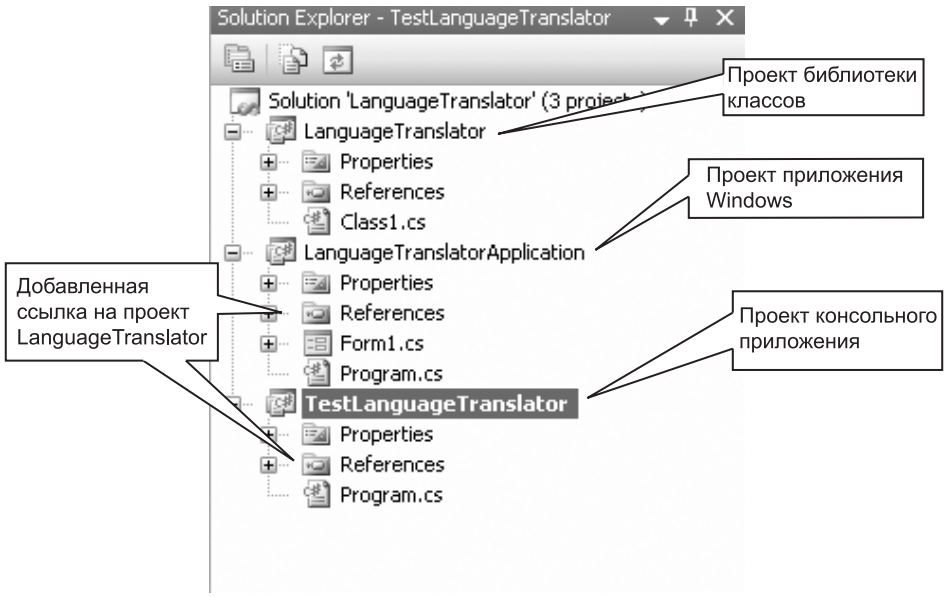

Рис. 3.1. Структура проектов приложения переводчика в проводнике решений Visual C# Express

Построение приложения переводчика

Приложение перевода, подобно приложению калькулятора в примере предыдущей главы, построено из нескольких частей: библиотека классов, выполняющая перевод на основании данных, предоставленных пользовательским интерфейсом, проверки и пользовательского интерфейса. Каждый фрагмент — это *компонент*. Сложенные вместе с другими компонентами, как в игре пазлы, они образует приложение.

На заметку. Компоненты — это основная часть вашей панели разработки. Как вы увидите, компоненты позволяют многократно инкапсулировать и использовать функциональные возможности. Компоненты существенно облегчают поддержку и расширение приложений. Конечно, существуют пределы, а преимущества не сами собой разумеющиеся. Чтобы извлечь пользу из использования компонентов, нужно правильно разработать свое приложение.

Создание класса `Translator`

При работе со средой разработки Visual C# Express или любым другим продуктом среды Visual Studio использование для создания библиотеки классов заданных по умолчанию шаблонов приводит к получению файла по имени `Class1.cs`. Конечно, хорошо, что для библиотеки классов создается стандартный файл, но идентификатор `Class1`.

cs подразумевает немного. Следовательно, вы должны удалить этот файл из проекта. Вместо него создадим класс Translator следующим образом.

1. Щелкните правой кнопкой мыши на проекте LanguageTranslator.
2. Выберите пункт меню Add⇨New Item (Добавить⇨Новый элемент).
3. Выберите Class (Класс).
4. Переименуйте файл в **Translator.cs**.
5. Щелкните на кнопке Add (Добавить), чтобы создать файл и добавить его в ваш проект.

Обратите внимание, как быстро вы сумели создавать класс C#, используя IDE Visual Studio. Легкость, с которой вы можете создавать файлы классов, позволяет сосредоточиться на добавлении в файл исходного кода. Однако не стоит впадать в заблуждение, что, создав ряд файлов классов, вы автоматически получите шедевр работоспособного кода. Вам нужно тщательно обдумать, какие файлы проектов, классов и проверок необходимо создать.

Перевод слова Hello

Первая возможность, которую мы реализуем, — это перевод текста “hello”. Поскольку слово “hello” на английском языке, первый перевод будет с английского на немецкий. Приведенный ниже код реализует эту возможность. Добавьте его в файл Translator.cs проекта LanguageTranslator.

```
public class Translator {
  public static string TranslateHello(string input)
  {
    if (input.CompareTo("hello") == 0)
    {
      return "hallo";
    }
    else if (input.CompareTo("allo") == 0)
    {
      return "hallo";
    }
    return "";
  }
}
```

Основной класс, Translator, предоставлен другим компонентам или частям исходного кода. Считайте его именем черного ящика. Черный ящик имеет один метод, TranslateHello(), который используется для преобразования французского слова *allo* и английского *hello* в немецкое слово *hallo*. Исходная переменная метода input имеет тип string, объект ссылочного типа.

В реализации метода TranslateHello() мы используем функцию CompareTo() для сравнения содержимого входного буфера с параметром "hello". Если они совпадают, значит строки равны и возвращается значение 0. Как будет изложено в разделе “Исследуем тип string”, строка — это объект, а объекты обычно имеют методы. Одним из методов типа string и является CompareTo().

Вызывающая сторона метода TranslateHello() не знает, как вы сумели преобразовать одно слово в другое. Это вызывающую сторону не заботит, ее заботит только то, чтобы метод вел себя так, как ожидается.

Абстрактно говоря, задача метода TranslateHello() в том, чтобы принять некоторый текст и, если он распознан, вернуть немецкое слово “hallo”.

Создание проверочного приложения

Безусловно, чтобы реализовать абстрактное намерение, мы должны проверить написанный код. Проверочный код добавляется в проверочное приложение, проект `TestLanguageTranslator`.

Добавьте в файл `Program.cs` следующий код.

```
static void TestTranslateHello()
{
  string verifyValue;

  verifyValue = LanguageTranslator.Translator.TranslateHello("hello");
  if (verifyValue.CompareTo("hallo") != 0)
  {
    Console.WriteLine("Test failed of hello to hallo");
  }

  verifyValue = LanguageTranslator.Translator.TranslateHello("allo");
  if (verifyValue.CompareTo("hallo") != 0)
  {
    Console.WriteLine("Test failed of allo to hallo");
  }

  verifyValue = LanguageTranslator.Translator.TranslateHello("allosss");
  if (verifyValue.CompareTo("") != 0)
  {
    Console.WriteLine("Test to verify nontranslated word failed");
  }

  verifyValue = LanguageTranslator.Translator.TranslateHello(" allo");
  if (verifyValue.CompareTo("hallo") != 0)
  {
    Console.WriteLine("Test failed of extra whitespaces allo to hallo");
  }
}
```

Исходный код содержит четыре проверки. Каждая вызывает метод `TranslateHello()` с определенными данными и получает результат. Проверка происходит при сравнении результата с ожидаемым значением. Для верификации правильности перевода используется функция `CompareTo()`.

Обратите внимание на третью проверку.

```
verifyValue = LanguageTranslator.Translator.TranslateHello("allosss");
if (verifyValue.CompareTo("") != 0)
  {
    Console.WriteLine("Test to verify nontranslated word failed");
```

Эта проверка явно не пройдет. Необходимо писать успешные проверки, которые не проходят. Успешные проверки, которые, как предполагается, потерпят неудачу, позволяют удостовериться, что ваш код не создает ошибочных срабатываний. *Ошибочное срабатывание* (false positive) — это, когда ваш код должен потерпеть неудачу, но не делает этого.

Проверку внутри метода должен вызвать метод `Main()`, как в следующем примере:

```
static void Main(string[] args)
{
  TestTranslateHello();
}
```

Если вы откомпилируете и запустите проверки, то обнаружите, что одна из проверок терпит неудачу. Неудачная проверка — четвертая, которая пытается перевести пустое слово. Отступ (один или несколько пробелов) — это то, на что люди обычно не обращают внимания, но от слов, предложений и прочего их следует отличать. Прежде чем мы сможем решить проблему отступа, необходимо определить, которая часть приложения работает неправильно.

Ответ на вопрос об ответственности

Проблема отступа весьма занимательна. Вызывающая сторона явно передает дополнительные пробелы, но действительно ли отступ — это ошибка или это данные, которые переданы по ошибке?

Проблема может быть определена в контексте ответственности применения. Предположим, что вы только что купили роскошный седан и ведете его по скоростной дороге. Если автомобиль развалится, наступит гарантийный случай и стоимость восстановления автомобиля будет фактически покрыта. Но если вы будете гонять автомобиль по проселкам с бревнами, выбоинами и камнями на каждом шагу, причем на высоких скоростях, то вряд ли отремонтируете его потом по гарантии.

Вернемся к компоненту перевода, он предоставляет метод `TranslateHello()` и несет ответственность за него. Вызывающая сторона метода `TranslateHello()` также несет ответственность за то, что передается ей для перевода. Таким образом, не вызывающая ли сторона несет ответственность за передачу отступа?

Если отступ — обычное дело, то неудача проверки — ошибка компонента. Но если отступ — ненормальное обстоятельство, то не права вызывающая сторона, ее и нужно исправить. И наоборот, если вызывающая сторона права, то в компоненте есть ошибка, которую следует устранить. Но как узнать, не вызывающей ли стороны это ответственность? Я поступил так потому, что именно так было в соглашении. Главное, определить хорошее соглашение.

Прежде чем я объясняю, как устранить ошибку, я должен вернуться назад и поговорить о том, что представляет собой строка и каковы ее возможности.

Исследуем тип `string`

Строка (string) — это объект, а следовательно, ссылочный тип. Тип `string` имеет методы и свойства. Типы значений, такие как `double` и `int`, тоже имеют методы и свойства, но `string` — это первый из рассматриваемых типов, который на самом деле является объектом, поэтому давайте исследуем его подробнее.

Чтобы исследовать тип, вы можете читать документацию или воспользоваться средством IntelliSense. Чтение документации — неплохая идея, но это долго и трудно. IntelliSense — это средство интегрированной среды разработки (IDE), представляющее методы и свойства определенного типа легким для понимания способом.

В первое время использование IntelliSense может немного нервировать, поскольку он, кажется, живет собственной жизнью, независимо от IDE. Небольшая инструкция по использованию IntelliSense приведена на рис. 3.2. Я рекомендую вам уделить некоторое время экспериментам с IntelliSense. Я также рекомендую вам включить IntelliSense в IDE Visual C# Express.

IntelliSense работает только для определенных переменных, имеющих тип. На рис. 3.2 средство IntelliSense сработало потому, что среда IDE анализирует код и способна читать метаданные, связанные с типом. *Метаданные* (metadata) — это данные, которые описывают ваш исходный код. Мы создаем его всякий раз, когда определяем класс, ме-

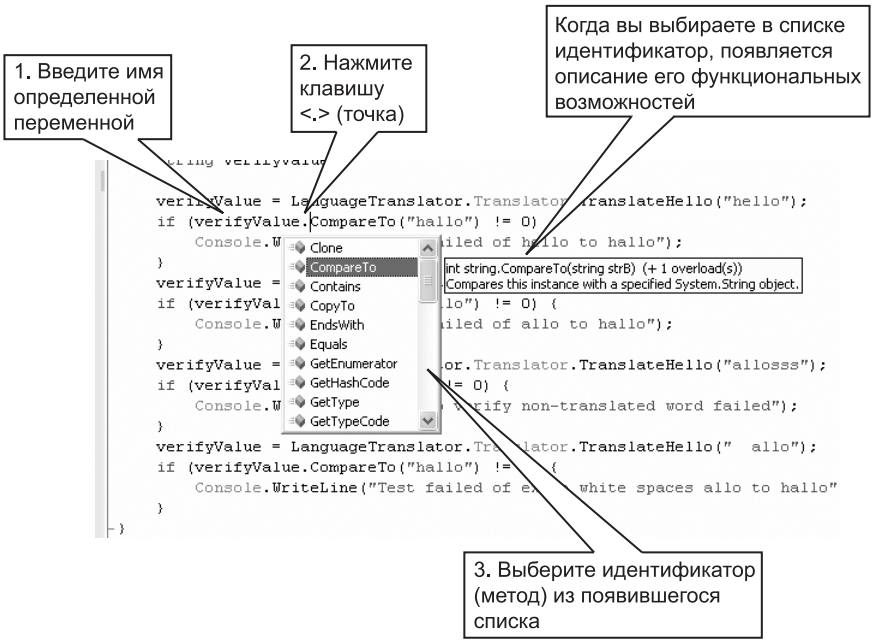

Рис. 3.2. Применение IntelliSense для переменной типа `string`

тод или свойство. Описания методов и свойств — это часть метаданных, отображаемых IntelliSense. Фактически одним из преимуществ платформы .NET является то, что метаданные имеют все типы.

Объект — основа всех типов

Практически все в .NET является объектами, обладающими несколькими базовыми свойствами и методами. Вот четыре базовых метода, связанных с каждым объектом.

- `Equals()`. Проверяет равенство двух объектов (рис. 3.3).
- `GetHashCode()`. Возвращает уникальное число, описывающее объект (рис. 3.4). Два объекта с одинаковым содержимым возвратят одинаковый хеш-код.
- `GetType()`. Возвращает метаданные, связанные с объектом (рис. 3.5). Позволяет программе динамически выяснять, какие методы и свойства какому типу принадлежат. Это используется IntelliSense, чтобы отобразить обычный список.
- `ToString()`. Преобразует содержимое типа в строку (рис. 3.6). Обратите внимание, стандартная реализация CLR метода `ToString()` работает только для типов значений.

Четыре базовых метода можно применить к любой объявленной переменной. Вы используете метод `ToString()` при отладке или проверке состояния экземпляра объекта во время выполнения. Метод `ToString()` возвращает понятную человеку строку, которая содержит состояние экземпляра объекта.

Вы можете использовать метод `GetType()` время от времени, но IDE и другие инструменты используют его регулярно. Этот метод позволяет выяснить возможности переменной во время выполнения программы. На техническом языке, метод `GetType()` возвращает формальное описание метаданных типа.

Из чтения описаний методов `Equals()` и `GetHashCode()` вы могли бы предположить, что эти две функции преследуют одну и ту же цель. Но дело обстоит не так.

Предположим, вы сложили и упаковали в две коробки кухонную утварь. В обеих коробках по пять красных блюд, три серебряных вилки, два медных ножа и два бокала. Если вы сравните коробки, методы `Equals()` и `GetHashCode()` укажут на равенство, коробки содержат одинаковое количество предметов одинаковых цветов. Важно уяснить, что, несмотря на совпадение предметов в двух коробках, они являются уникальными экземплярами, содержащими уникальные предметы, хоть их содержимое и идентично. Когда вы сравниваете экземпляры объектов с помощью методов `Equals()` или `GetHashCode()`, вы сравниваете метаданные и оцениваете атрибуты, а не индивидуальные экземпляры.

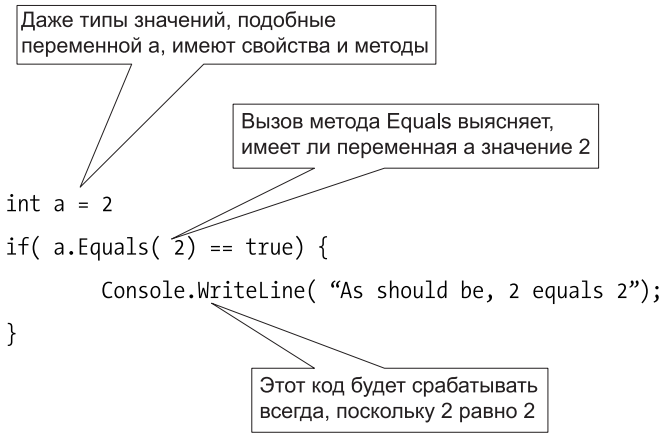

Рис. 3.3. Метод `Equals()` используется для проверки равенства двух объектов

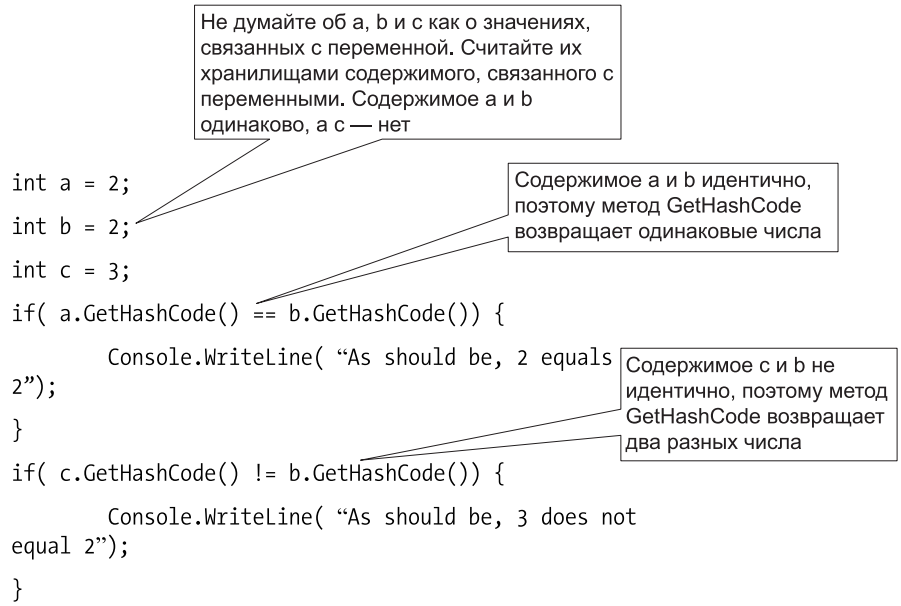

Рис. 3.4. Метод `GetHashCode()` возвращает уникальное число, описывающее объект

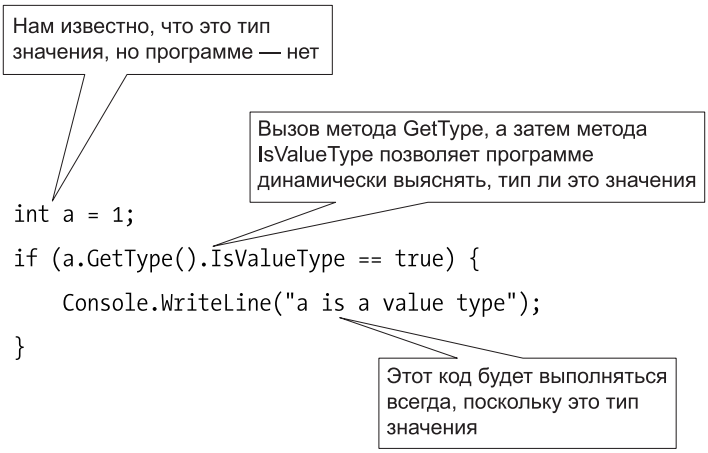

Рис. 3.5. Метод `GetType()` возвращает метаданные, связанные с объектом

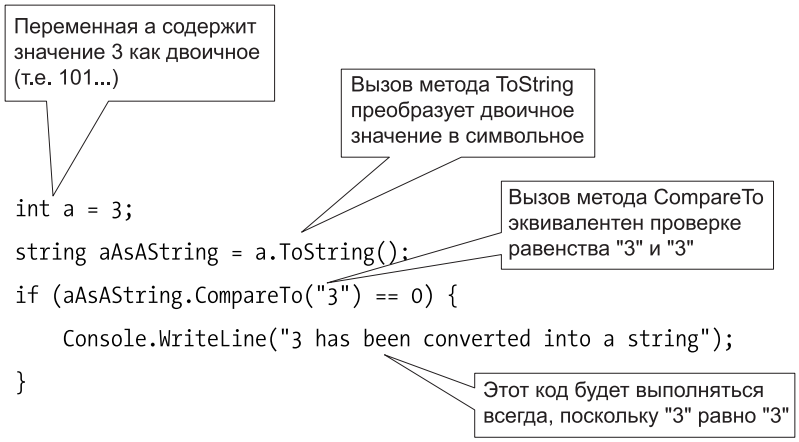

Рис. 3.6. Метод `ToString()` преобразует содержимое типа в строку

Теперь представьте, что бокалы в одной из коробок от IKEA, а в другой — от Pier 1. Если для сравнения коробок вы используете метод `Equals()`, он возвратит значение `false`, поскольку в подробностях содержимое коробок не идентично. Различие заключается в описании бокалов. Вызов метода `GetHashCode()`, напротив, укажет, что содержимое коробок идентично. Это связано с тем, что метод `GetHashCode()` осуществляет быструю идентификацию содержимого.

Различие между методами `Equals()` и `GetHashCode()` в точке зрения. С точки зрения транспортной компании, коробки идентичны, поскольку ее не заботит, лежат ли там бокалы от IKEA или от Pier 1; перевозчикам безразличен производитель бокалов.

Тот факт, что метод `GetHashCode()` может вернуть одинаковые числа для объектов с, казалось бы, несходным содержимым, может запутать разработчиков. Преимущество метода `GetHashCode()` в том, что, полезный при проверке равенства, он еще более полезен при проверке *неравенства*. Если два объекта возвращают разные значения хеш-

кода, то вы знаете, что содержимое не идентично. Назначение хеш-кода в том, чтобы быстро снять отпечатки пальцев содержимого объекта. Это не абсолютно надежно, но срабатывает обычно быстро.

Когда IntelliSense не достаточно

Система IntelliSense очень полезна, она отображает даже комментарии, объясняющие, что делает метод (см. рис. 3.2). Еще одно место для поиска ответов — это сама документация Microsoft, к которой можно обратиться, выбрав пункт меню Help⇨Index (Справка⇨Индекс). Чтобы поискать определенный тип, вы можете использовать поле Look For (Поиск). Например, если вы введете в поле поиска **`String class`**, то увидите подробности о классе `String`, которые можно отфильтровать, используя ссылки верху страницы.

Документация Microsoft — это часть библиотеки *сети разработчиков Microsoft* (Microsoft Developer Network — MSDN) на веб-сайте `http://msdn.microsoft.com`. Веб-сайт MSDN содержит документацию, которая поможет вам изучить *интерфейс прикладных программ* (Application Programming Interface — API) стандартного *комплекта разработчика программного обеспечения* (Software Development Kit — SDK) .NET. Существуют буквально тысячи типов, предоставляющих свои методы и свойства. Вряд ли вы используете их все в одном приложении, но SDK .NET вы будете использовать всегда.

В большинстве случаев веб-сайт MSDN вполне способен предоставить всю необходимую информацию о любом типе. Но если вы хотите узнать больше, то просмотрите такие веб-сайты, как Code Project (`http://www.codeproject.com`). Сайт содержит множество примеров по практически каждой теме разработки, которая может прийти на ум.

Проблема: посимвольное сравнение

Давайте вернемся к ошибке отступа. Проблемы создавал метод `CompareTo()`. Рассматривая документацию MSDN, вы можете найти следующее определение для этого метода (достаточно прокрутить страницу класса `String` вниз и щелкнуть на ссылке CompareTo).

Сравнивает данный экземпляр с определенным объектом.

Это определение говорит вам немного, так что вам нужно щелкнуть на имени другого метода под заголовком Reference (Ссылка). (Зачастую объявление другого метода, доступного по ссылке, содержит объяснение общих концепций.) Щелкните на ссылке метода `Compare()`, а затем `Compare(string, string)`. В объяснении метода `Compare()` вы найдете следующий текст.

Сравнение завершается, когда обнаруживается первое неравенство символов или обе строки заканчиваются. Однако, если обе строки равны до конца одной из строк, но в другой строке символы еще остались, то более длинная строка считается большей. Возвращаемое значение — результат сравнения, выполняемого последовательно.

На заметку. Поиск описания метода — процесс недолгий, даже если он находится в описании другого метода. Накопив некоторый опыт, вы даже перестанете обращать внимание на дополнительные щелчки.

Метод `CompareTo()` терпит неудачу из-за посимвольного сравнения, которое проиллюстрировано на рис. 3.7.

Как вы знаете, строки хранятся в областях памяти, называемых *буферами* (buffer), где каждый пробел занимает один символ (см. рис. 3.7). Иногда это можно использовать, как будет показано в следующем разделе.

Теперь, когда вы знаете, в чем проблема, пришло время искать решение.

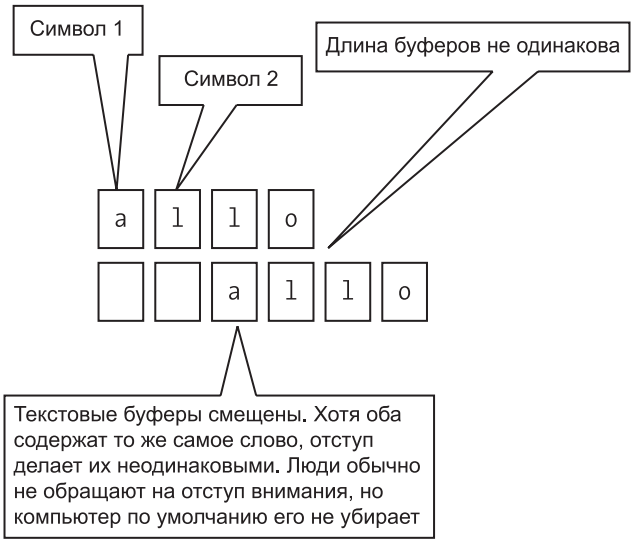

Рис. 3.7. Вот почему метод `CompareTo()` считает неодинаковыми строки, кажущиеся идентичными, но имеющие дополнительные символы

Решение проблемы отступа

Можно решать проблему отступа разными способами. Используемый подход будет зависеть от ваших потребностей. Давайте рассмотрим несколько решений и выясним, какой из них лучше подходит для нашей программы перевода.

Обрезка отступа

Первое решение заключается в удалении отступа с помощью метода, специально предназначенного для этой цели. Другими словами, проблема отступа не уникальна и фактически хорошо известна. Тип `string` имеет метод, применяемый для обрезки, или удаления, отступа из буфера. Вы можете удалять отступ в начале, в конце или с обеих сторон буфера.

Как ни соблазнительно изменить исходную реализацию метода `TranslateHello()`, не торопитесь делать это, поскольку вы можете что-нибудь нарушить. Разрабатывая код, вы имеете несколько возможных способов решения проблемы. Если вы не будете серьезно относиться к первоначальному исходному коду, то ко времени третьего или четвертого решения код может оказаться в полном беспорядке. Ваши исправления могут все испортить, а вернуться к первоначальному коду будет уже затруднительно.

На заметку. Для обслуживания своего исходного кода вы должны использовать *контроль версий* (version control). Но даже при контроле версий, когда вы удаляете прошлые попытки, идеи можно потерять. Таким образом, несмотря на чистоту исходного кода, вы можете забыть то, что делали три или четыре часа назад. Поверьте мне, это случается, поскольку разработка исходного кода — процесс, требующий больших умственных усилий.

Решение заключается в создании *прокладки* (shim), которая вызывает метод `TranslateHello()`. В механике прокладку помещают между двумя другими деталями, чтобы улучшить соединение. В программировании прокладка — это код, помещаемый

между двумя другими слоями кода, чтобы устранить ошибку и повысить эффективность кода. В данном случае наша прокладка будет использоваться для устранения ошибки. Следующий код прокладки — это временное решение.

```
public static string TrimmingWhitespace (string buffer)
{
  string trimmed = buffer.Trim();
  return LanguageTranslator.Translator.TranslateHello(trimmed);
}
```

`TrimmingWhitespace()` — это временный метод, который вырезает отступ из строки, передаваемой на перевод; `buffer.Trim()` — это новая функция, предварительно обрабатывающая буфер. И наконец, происходит вызов первоначального метода, `TranslateHello()`, осуществляющего перевод.

Конечно, необходимо проверить новый метод, чтобы убедиться, обрезает ли он передаваемую строку. Соответствующий проверочный код приведен ниже.

```
verifyValue = TrimmingWhitespace(" allo");
if (verifyValue.CompareTo("hallo") != 0)
{
  Console.WriteLine("Test failed of extra white spaces allo to hallo");
}
```

Проверка вызывает метод `TrimmingWhiteSpace()`, чтобы убедиться, все ли сработало. Код верификации не изменился.

Итак, вы получили ожидаемый результат, но не забывайте, что вызов прокладки — это не первоначальный метод. Если вы запустите проверочный код, то убедитесь, что прокладка работает и, таким образом, решение есть.

Поиск подстроки

Другое решение проблемы отступа — поиск в буфере определенной подстроки. Решение подразумевает перебор строки в поисках соответствия элемента буфера некоторому тексту, который ищут. Временный рабочий код приведен на рис. 3.8.

Проверочный код здесь не приведен, поскольку он такой же, как в предыдущем решении; различие лишь в проверяемом методе.

Какое решение лучше?

Уделите момент размышлениям о том, какое решение лучше — обрезка отступа или поиск подстроки? Ни одно из них не совершенно; каждое имеет свои недостатки. Как вы уже, возможно, знаете, это обычное дело при разработке программного обеспечения. Вы думаете, что заткнули все дыры, но при новых обстоятельствах ваше программное обеспечение снова не работает. Случалось ли с вами такое? Подозреваю, что да.

Снова хочу подчеркнуть, что вам понадобится написать больше проверок, чтобы выяснить, какие случаи могли бы нарушить работу программного обеспечения. Для решения обрезки отступа приведенная ниже проверка приведет к отказу.

```
verifyValue = TrimmingWhitespace("a allo");
if (verifyValue.CompareTo("hallo") != 0)
{
  Console.WriteLine("Test failed: cannot parse multiple words");
}
```

В этой проверке первая буква “a” окажется первым символом и не будет удалена. Верификация потерпит неудачу, поскольку метод `CompareTo()` не сможет сравнить смещенный буфер из-за начального символа “a”.

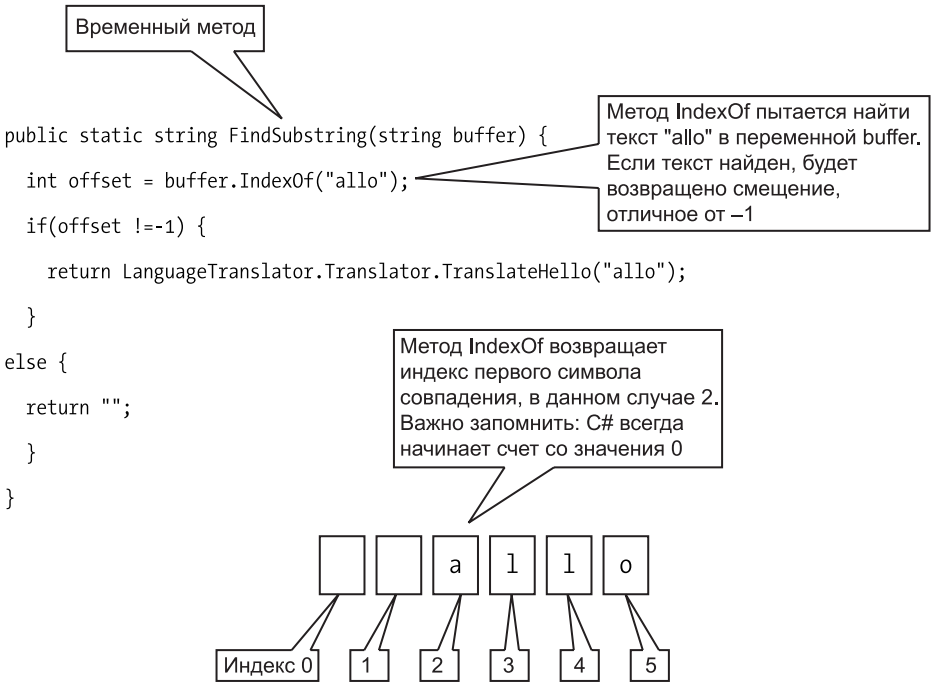

Рис. 3.8. Решение проблемы отступа за счет поиска подстроки

Если бы новая проверка была применена к решению подстроки, она прошла бы успешно, и слово `allo` оказалось бы найдено. Поскольку новая проверка заставила первое решение потерпеть неудачу, на этом решении, казалось бы, и стоит остановиться. Наше доверие ко второму решению увеличилось потому, что старая и новая проверки прошли удачно. Но, возможно, мы не должны настолько доверять поспешным суждениям. Решение подстроки терпит неудачу при следующей проверке.

```
verifyValue = FindSubstring("allodium");
if (verifyValue.CompareTo("hallo") != 0)
{
  Console.WriteLine("Test failed: cannot parse multiple words");
}
```

Проверяемое слово, `"allodium"`, содержит символы `allo`. Верификация успешна — это хороший пример ошибочного срабатывания.

На заметку. Как можно заметить, важно иметь побольше проверок, которые проверяют множество разных сценариев. Удостоверьтесь также, в наличии проверок, которые должны быть пройдены, и проверок, которые должны терпеть неудачу.

Подведем итог, ни одно из решений не работает правильно. Дополнительная проверка каждого из решений выявила новые проблемы. Поэтому вполне очевидно, что необходимо искать другое решение.

Сложности разработки

Казалось бы, что выработка решений и последующее создание аннулирующих их проверок — это упражнение по самобичеванию. Вы многократно создаете код, который не решает проблемы. Если вы последовательны в своих проверках, считайте это частью разработки программного обеспечения. Некоторые разработчики пишут код и не волнуются о проверках, такие разработчики создают разработке программного обеспечения плохую репутацию. Если вы хотите быть разработчиком, заслуживающим доверия, кто-то должен проверять ваш код.

Создание проверок до разработки кода

Небольшой Дзен о проверке: причина, по которой последующее решение терпит неудачу, в том, что каждое последующее решение было сродни безусловному рефлексу. Безусловный рефлекс — когда программист, столкнувшись с ошибкой, устраняет только ее — не больше и не меньше. Вместо этого программист должен выяснить, в чем причина ошибки, и попытаться устранить ее. Первоначальная ошибка с отступом не была ошибкой отступа, а ошибкой, говорящей: "Эй, а если начало текста или предложения не совпадают?"

Поэтому для исправления ошибки вы должны не только написать код; вы должны продумать все проверки, которые ваш код должен пройти. Нужно назначить ответственного и определить критерии успеха и провала. В примере перевода подходящий подход реализации должен был бы подразумевать написание проверок перед написанием кода. В табл. 3.1 приведены критерии успеха и неудачи.

Таблица 3.1. Подходящие проверки для программы перевода

Проверка	Результат верификации
`allo`	Успех
`"allo"`	Успех
`word allo`	Неудача. Вы не можете перевести одно слово без перевода других
`word allo word`	Неудача. Та же причина, что и у слова `allo`
`prefixallo`	Неудача. Другое слово
`alloappend`	Неудача. Другое слово
`prefixalloappend`	Неудача. Другое слово

Как можно заметить, большинство случаев проверки — отказы, поскольку компонент перевода, как предполагается, переводит только отдельные слова. Набор проверок кажется законченным, но фактически мы пропустили еще один набор случаев, который приведен в табл. 3.2.

Таблица 3.2. Пропущенный набор проверок для программы перевода

Проверка	Результат верификации
`Allo`	Успех
`"allO"`	Успех

Текст может содержать символы в разном регистре, с точки зрения человека, это то же самое слово (даже при том, что его можно посчитать опечаткой). Но компьютер рассматривает такой случай как совершенно иной буфер, и нам следует справиться с этой ситуацией.

На рис. 3.9 показано рабочее решение.

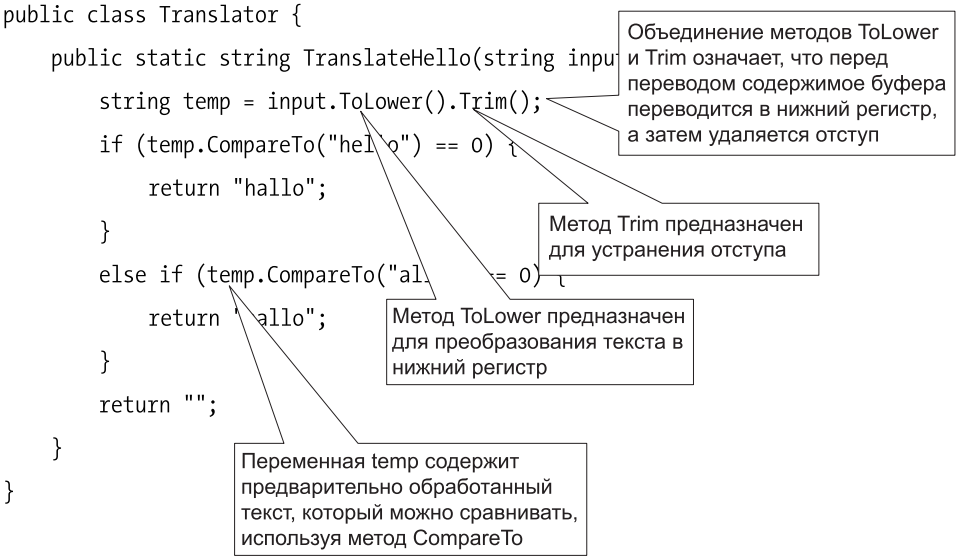

Рис. 3.9. Окончательное решение перевода

Рассмотрев это решение, вы обратите внимание на элементы, которые уже были в первом решении. Но это решение было не особенно ценно, поскольку оно не работало правильно. Это пример того, как поиск быстрого решения только для ошибки, сбойного кода, в отрыве от решения проблемы, может завести в тупик. Я знаю, что слишком далеко отклоняюсь от темы, но полагаю, что это важно. Чтобы стать настоящим разработчиком программного обеспечения, вы должны думать не только о том, почему нечто не работает, и не только устранять ошибки, чтобы избавиться от них.

На заметку. Все решения используют методы типа `string`. Тип `string` очень сложен и поддерживает множество действий, обычно выполняемых с текстом.

В настоящий момент мы закончим с переводом приветствия. Впоследствии я укажу пару дополнительных элементов, которые необходимо иметь в виду, работая со строками.

Строка в кавычках

Вы, возможно, уже обратили внимание на использование парных и одиночных кавычек при вызове метода `CompareTo()`. Между типами используемых кавычек есть существенное различие. Если вы используете двойные кавычки, как в следующем примере, то вы указываете строковый тип:

```
"using double quotes"
```

Если вы используете одинарные кавычки, как в следующем примере, то вы указываете символ:

```
'a'
```

Одинарные кавычки применимы только к отдельным символам. Считайте одиночный символ буквой. Однако не стоит слишком полагаться на это определение, поскольку не все языки используют буквы. Если вы попытаетесь использовать одинарные кавычки для помещения в буфер нескольких символов, то компилятор C# выдаст сообщение об ошибке, обычно называемое в .NET *исключением* (exception)[1].

Соотнесение символов

Один символ занимает 16 битов, а место, занимаемое строкой, зависит от количества символов в буфере. Если буфер содержит десять символов, то весь буфер займет 160 битов. Не забывайте, что тип `string` ссылочный, а не тип значения.

Один символ имеет длину 16 битов, так что буфер может хранить текст в множестве разных форматов. Стандартная длина определена стандартом Unicode.

Рассмотрим символ *a*. С философской точки зрения, откуда вы знаете, что *a* — это *a*? Для людей все просто, поскольку их мозг с детства приучен распознавать эту кривулину как *a*. Теперь рассмотрим русскую букву, представленную на рис. 3.10.

Рис. 3.10. Русская буква

Какая буква изображена на рис. 3.10? Выглядит похоже на *H*, правильно? Но в английском языке этой букве соответствует *N*. Русский язык имеет собственный набор символов, и кто-то определил[2], что русская *H* — это английская *N*. На рис. 3.11 показано соотнесение русских и английских букв.

Если бы я изучал русский язык, то использовал бы соотнесение, приведенное на рис. 3.11. Это позволило бы мне получить представление о каждой русской букве. Вы можете считать рис. 3.11 таблицей подстановок. Компьютер имеет ту же потребность, поскольку он не понимает символы. Компьютер понимает только числа, а используя таблицы подстановок, он соотносит наборы символов с наборами чисел.

Существует множество таблиц подстановок, в частности *американский стандартный код для обмена информацией* (American Standard Code for Information Interchange — ASCII). Например, в таблице ASCII символу *a* соответствует число 97. Проблема стандарта ASCII в том, что он хорош для английского языка, но не для других языков. Стандарт ASCII расширен для западноевропейских языков, но все еще не подходит для таких языков, как китайский, русский[3] и арабский.

Решение, выбранное .NET, — это Unicode. Unicode — определение набора таблиц подстановок, которые сопоставлены символами для всех языков мира.

---

[1] Исключение — это скорее полнофункциональный объект, одним из свойств которого является текст сообщения. — *Примеч. ред.*

[2] Братья Кирилл и Мефодий. См. `http://www.zn.ua/3000/3150/46511/`. — *Примеч. ред.*

[3] Без комментариев. — *Примеч. ред.*

Russian	English transliteration	Russian	English transliteration
а	a	р	r
б	b	с	s
в	v	т	t
г	g	у	u
д	d	ф	f
е / ё	e	х	kh
ж	zh	ц	ts
з	z	ч	ch
и / й	i	ш	sh
к	k	щ	shch
л	l	ъ	´´
м	m	ы	y
н	n	ь	´
о	o	э	e
п	p	ю	iu
Sennaya.com		я	ia

Рис. 3.11. Соотнесение русских и английских букв

Как правило, вам не придется самостоятельно работать с Unicode, поскольку среда .NET все сделает сама. Не так давно программисты были вынуждены сами управляться с таблицами подстановок. Поэтому, если вы новичок в программировании, то считайте, что вам повезло и вы избежали головной боли при разработке многоязычных приложений.

Работа с языками и национальными форматами

Управление строками в среде .NET не останавливается на Unicode. Среда .NET весьма творчески подходит к этому вопросу, она понимает такие концепции, как национальный формат и язык, отражающие то, как люди говорят и живут. Концепций национального формата и языка нет в других средах программирования.

Рассмотрим Швейцарию, которая размером со штаты Вермонт и Нью-Гемпшир вместе. Горы разделяют Швейцарию на четыре региона, каждый из которых имеет собственный язык: немецкий, итальянский, ретороманский и французский. Несмотря на наличие четырех языков, все в Швейцарии используют одинаковую валюту и форму записи чисел.

В предыдущих средах программирования язык был привязан к специфической стране. Это прекрасно работает для Франции, Германии и Соединенных Штатов, но не подходит для Канады, Швейцарии, Бельгии и Индии. Язык необходимо отделить от национального формата, поскольку некоторые культуры используют по несколько языков. На итальянском, например, разговаривают в Швейцарии и Италии. На французском языке разговаривают во Франции, Швейцарии, Люксембурге и Канаде. На немецком — в Германии, Швейцарии и Австрии.

Установка языка и национального формата в Windows

Операционная система Windows позволяет установить национальный формат и язык вашего компьютера, независимо от языка, на котором действует Windows. Пример приведен на рис. 3.12.

Пример на рис. 3.12 использует немецкую версию Windows для Швейцарии. Язык английский, а национальный формат канадский. Казалось бы, операционная система Windows должна запутаться, но фактически, если вы пишете приложение .NET правильно, многоязычная поддержка очень проста.

Анализ и обработка чисел

Национальный формат и страна будут очень важны, когда дело дойдет до взаимодействия с числами и датами, хранящимися в виде строк. Представьте получение строкового буфера со встроенным числом и последующую попытку выполнения сложения, как проиллюстрировано на рис. 3.13.

Сумма чисел — это, конечно, математическая операция. Когда операция сложения выполняется для строк, она всегда приводит к конкатенации буферов. Суммирование — самый простой способ конкатенации строковых буферов.

Однако конкатенация — это не цель примера. Его задача в том, чтобы превратить строки в числа, а затем сложить их, чтобы переменная `c` содержала значение `3` (1 + 2 = 3). Переделанная версия примера приведена на рис. 3.14. Этот код анализирует строку как целое число.

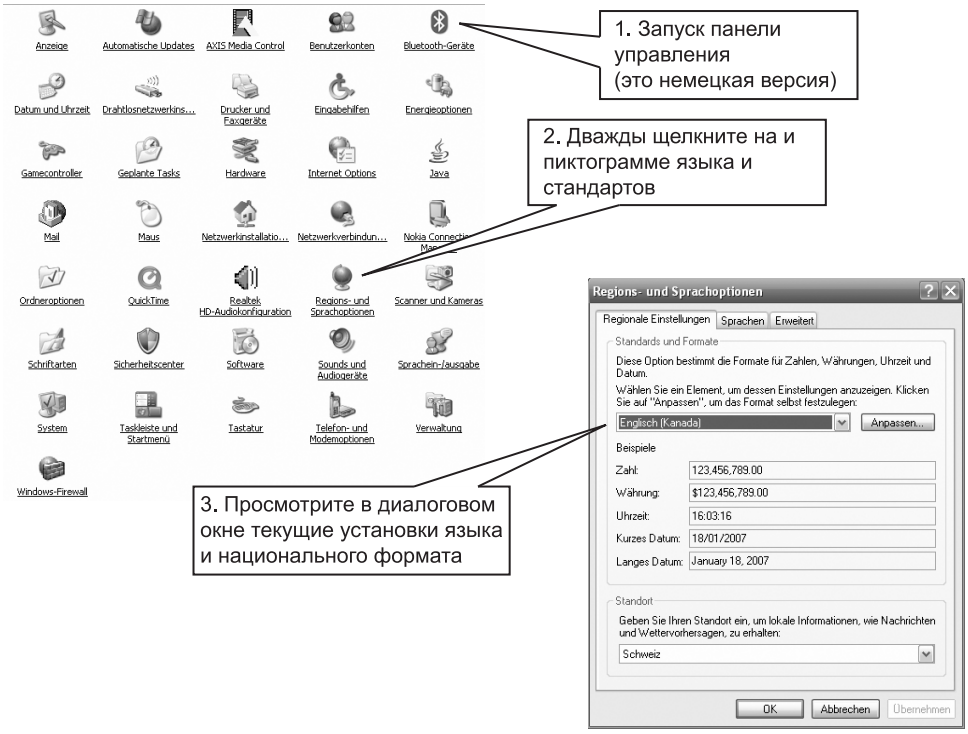

Рис. 3.12. Установка языка и национального формата в Windows

Рис. 3.13. Арифметика с числами, представленными строками, приводит к непредвиденным результатам

Тип `int` обладает методом `Parse()`, применяемым для преобразования строки в целое число. Анализ срабатывает, только если буфер содержит допустимое число. Если буфер содержит символы или недопустимое число, произойдет ошибка.

Чтобы код мог справиться с неудачным преобразованием строк, решение, используемое функциями анализа, заключается в создании исключения, которое программа могла бы обработать. В качестве альтернативы отказоустойчивый способ анализа чисел без блока исключения подразумевал бы использование метода `TryParse()`, как в следующем примере.

```
int value;
if(int.TryParse("1", out value)) { }
```

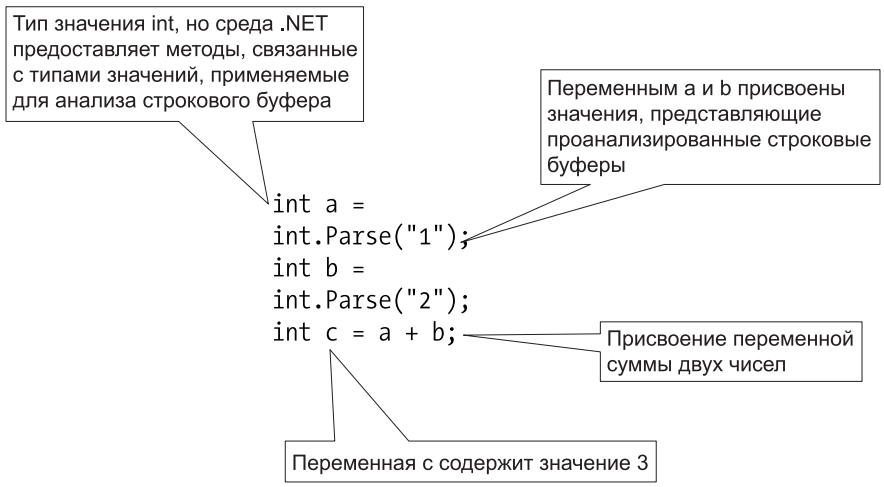

Рис. 3.14. Преобразование строк в целые числа перед выполнением арифметических операций

Метод `TryParse()` не возвращает целочисленного значения, он возвращает логический (`bool`) флаг (истина (`true`) или ложь (`false`)), указывающий на возможность анализа буфера. Если возвращено значение `true`, то буфер можно проанализировать, а результат сохранить в значении параметра, помеченного идентификатором `out`. Идентификатор `out` используется в среде .NET для указания на то, что параметр со-

держит значение после вызова метода `TryParse()`. Вы можете проанализировать числа других типов, используя те же методы (например, `float.TryParse()`).

Существует множество возможных вариантов анализа чисел. Например, число `"100"` может быть проанализировано как шестнадцатеричное. (Компьютеры могут использовать шестнадцатеричные числа. Шестнадцатеричная форма записи (hexadecimal) зачастую называется шестнадцатеричным кодом (hex) для краткости.) Вот пример шестнадцатеричного преобразования.

```
using System.Globalization;
 ...
public void ParseHexadecimal()
{
  int value = int.Parse("100", NumberStyles.HexNumber);
}
```

Этот пример использует тот вариант метода `Parse()`, который имеет второй, дополнительный параметр, представляющий формат числа. В данном случае второй параметр указывает, что формат числа шестнадцатеричный (`NumberStyles.HexNumber`, из пространства имен `System.Globalization`). В примере буфер представляет десятичное число 256, которое проверяется с помощью метода `Assert.AreEquals()`.

На заметку. Если вас интересует вопрос, как 100 превращается в 256 на шестнадцатеричном уровне, используйте калькулятор операционной системы Windows. Переведите калькулятор в инженерный вид, щелкните на переключателе Hex, введите число **100**, а затем щелкните на переключателе Dec.

Перечисление `NumberStyles` имеет другие значения, применяемые для анализа числа по другим правилам. Например, некоторые правила считают круглые скобки вокруг числа указанием на отрицательное значение. Другие правила относятся к отступам. Например.

```
public void TestParseNegativeValue()
{
  int value = int.Parse( " (10) ",
    NumberStyles.AllowParentheses |
    NumberStyles.AllowLeadingWhite |
    NumberStyles.AllowTrailingWhite);
}
```

Число[4] `" (10) "` в этом примере намеренно усложнено, оно имеет отступ и круглые скобки. Попытка анализировать число с помощью метода `Parse()` без применения значений перечисления `NumberStyles` не сработает. Значение перечисления `AllowParentheses` разрешает обрабатывать круглые скобки, `AllowLeadingWhite` — игнорировать предваряющие пробелы, а `AllowTrailingWhite` — завершающие. После обработки буфера в переменной `value` будет сохранено значение `-10`.

Другие значения перечисления `NumberStyles` позволяют учитывать десятичные точки дробных чисел, положительные или отрицательные числа и т.д. Это поднимает тему обработки чисел, отличных от типа `int`. Каждый из базовых типов данных (`boolean`, `byte` и `double`) имеет собственные версии методов `Parse()` и `TryParse()`. Кроме того, метод `TryParse()` также может использовать перечисление `NumberStyles`. (Более подробная информация о значениях перечисления `NumberStyles` приведена в документации MSDN.)

---

[4] Здесь и далее автор применяет слово “число”, подразумевая строковое значение, а не числовое. — *Примеч. ред.*

Анализ целочисленных значений осуществляется одинаково, независимо от страны, а вот анализ дат или типа `double` — нет. Рассмотрим следующий пример, в котором осуществляется попытка проанализировать буфер, содержащий десятичное значение.

```
public void TestDoubleValue()
{
  double value = double.Parse("1234.56");
        value = double.Parse("1,234.56");
}
```

В обоих случаях этого примера метод `Parse()` используется для преобразования числа 1234,56. В первом случае анализ прост, поскольку число содержит лишь десятичную точку, отделяющую целые значения от десятичных. Во втором случае анализ усложнен, поскольку число содержит запятую, отделяющую тысячи от целого числа. В обоих случаях метод `Parse()` сработает нормально.

Если вы проверите этот код, то, возможно, получите исключение. В этом случае виноват национальный формат приложения. Числа в примере предназначены для формата `en-CA`, английский Канада.

Работа с национальным форматом

В среде .NET информация о национальном формате реализована с использованием двух идентификаторов — *языка* (language) и *специализации* (specialization). Как я уже упомянул, в Швейцарии четыре разговорных языка, что означает также четыре разных способа выражения дат, времени и валют. Это не означает, что дата у немецкоязычных и франкоязычных людей разная. Формат даты будет одинаков, но слова *Maerz* и *Mars* в описании месяца марта будут различны. Слова для даты будут те же, что и для Австрии, Швейцарии и Германии, но формат будет другой. Это значит, что такие многоязычные страны, как Канада (французский и английский языки) и Люксембург (французский и немецкий языки) должны быть способны использовать несколько кодировок, а следовательно, нуждаются в двух идентификаторах.

Чтобы получить информацию о текущем национальном формате, используйте следующий код.

```
CultureInfo info =
  Thread.CurrentThread.CurrentCulture;
Console.WriteLine(
  "Culture (" + info.EnglishName + ")");
```

Метод `Thread.CurrentThread.CurrentCulture()` возвращает информацию о национальном формате, связанную с потоком, выполняющимся в настоящий момент. Как побочное явление, появляется возможность связать различные потоки с различной национальной информацией. Свойство `EnglishName` задает английскую версию информации о национальном формате. Это будет выглядеть следующим образом.

```
Culture (English (Canada))
```

Рассмотрим следующее число.

```
1,234
```

В американском или канадском национальном формате это число — одна тысяча двести тридцать четыре, но в немецком национальном формате — это одна целая, двести тридцать четыре тысячные (в Германии запятая используется как десятичный разделитель, а точка — как разделитель тысяч). Один из способов смены национального формата возможен в диалоговом окне, приведенном на рис. 3.12. Второй способ возможен на программном уровне, как в следующем коде.

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("en-CA");
```

В этом примере создается новый экземпляр объекта CultureInfo, содержащий информацию национального формата en-CA.

Следующий пример обрабатывает число типа double, используя кодировку немецкого формата.

```
public void TestGermanParseNumber()
{
  Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");
  double value = Double.Parse("1,234");
}
```

В этом примере текущему потоку присваивается информация о национальном формате de-DE. Теперь, при использовании любой из функций анализа, будут применяться правила немецкого форматирования для Германии. Изменение информации о национальном формате не повлияет на правила форматирования для языка программирования.

Используя функции Parse() и TryParse(), можно также анализировать дату и время, как показано в следующем примере.

```
public void TestGermanParseDate()
{
  DateTime datetime = DateTime.Parse("May 10, 2005");
  Assert.AreEqual(5, datetime.Month);
  Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");
  datetime = DateTime.Parse("10 Mai, 2005");
  Assert.AreEqual(5, datetime.Month);
}
```

Обратите внимание, как первый вызов метода DateTime.Parse() обрабатывает текст в английско-канадском формате и узнает, что идентификатор May эквивалентен пятому месяцу года. Перед вторым вызовом метода DateTime.Parse() национальный формат изменен на немецкий. В результате появилась возможность обработать строку 10 Mai, 2005. В обоих случаях обработка буфера не создала никаких серьезных проблем, поскольку вы знали, что буфер содержал дату в немецком или английско-канадском формате. Но все может стать куда сложней, когда дата будет немецкой, а национальный формат — английским.

Среда .NET 2.0 преобразует значение типа данных в содержимое буфера относительно просто, поскольку это осуществляет метод ToString(). Рассмотрим следующий пример, создающий буфер из целочисленного значения.

```
public void TestGenerateString()
{
  String buffer = 123.ToString();
  Assert.AreEqual("123", buffer);
}
```

В этом примере значение 123 было неявно преобразовано в переменную и фактически не представляет переменную 123. Затем для значения 123 можно применить метод ToString(), который создает буфер, содержащий строку "123". Та же самое можно проделать с числом типа double, как в этом примере.

```
double number = 123.5678;
String buffer = number.ToString("0.00");
```

Здесь число 123.5678 преобразовано в содержимое буфера с использованием метода ToString(), но у метода ToString() есть параметр, инструкция по форматированию, указывающая, как число типа double должно быть отображено в буфере. Желательный

результат — буфер с двумя цифрами после запятой. Поскольку третья цифра после десятичной запятой — 7, значение округляется до `123.57`.

Давайте рассмотрим пример, где информация о национальном формате применяется также к получаемому буферу. Здесь значение типа `double` создается в национальном формате.

```
public void TestGenerateGermanNumber()
{
  double number = 123.5678;
  Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");
  String buffer = number.ToString("0.00");
  Assert.AreEqual("123,57", buffer);
}
```

Как и в предыдущих примерах, свойство `CurrentCulture` применяется для указания желаемого национального формата. Затем происходит вызов метода `ToString()` для переменной `number` типа `double`, имеющей собственный метод `ToString()`, создающий буфер `"123,57"`.

Что следует запомнить

В этой главе вы изучили строки и написание кода. Вот основные моменты, которые стоит запомнить.

- Написание проверок — важнейшая часть практики разработки. Проверка — это не только механизм предотвращения ошибок, но и механизм, используемый для понимания динамики вашего кода.
- Тип `string` — это специальный ссылочный тип, имеющий множество методов и свойств. Чтобы узнать, на что способен тип, советую изучить документацию MSDN.
- Средство IntelliSense и документация MSDN — вот ваши помощники при поиске определенных методов, свойств или типов. Книги и веб-сайты, такие как Code Project, хорошие ресурсы для понимания концепций.
- Все переменные и типы основаны на типе `object`.
- При написании кода необходимо определить обязанности и контексты. Не устраняйте ошибки и не пишите код, подчиняясь условному рефлексу.
- Все строки основаны на стандарте Unicode. Каждый символ Unicode размером в 16 битов.
- При преобразовании буферов приходится иметь дело не только с преобразованием текста, но и чисел и дат.
- Среда .NET обладает сложной технологией, помогающей переводить числа и даты, используя комбинацию информации о языке и национальном формате.

Самостоятельные упражнения

Ниже приведено несколько упражнений, относящихся к изученному в этой главе.

1. Закончите приложение перевода с одного языка на другой, позволив пользователю самостоятельно выбрать направление перевода.
2. Дополните компонент `LanguageTranslator` так, чтобы появилась возможность переводить слова *au revoir* и *auf wiedersehen* в *good bye*.

3. Вы можете объединять строки, используя знак “плюс”, но большое количество конкатенаций замедляет код. Для конкатенации двух буферов используйте класс `StringBuilder`. Подсказка: вы хотите переделать код `string c = a + b` с помощью класса `StringBuilder`. Результат должен быть присвоен переменной `c`.
4. Создайте проверку, демонстрирующую, что происходит, когда числовое значение добавляется к строковому. Напишите соответствующие проверки, чтобы подтвердить свое заключение.
5. Дополните компонент `LanguageTranslator` методами перевода чисел в американском формате на немецкие.
6. Дополните компонент `LanguageTranslator` методами перевода американских и канадских дат в немецкие. Обратите внимание, это добавление позволит вводить американские и канадские даты.
7. Реализуйте приложение Windows, обращающееся к компоненту `LanguageTranslator`.