# 17

# Локализация

Космическая станция Mars Climate Orbiter, стоимостью 125 миллионов долларов, была потеряна 23 сентября 1999 года потому, что в одной из ответственных операций одна команда инженеров использовала единицы измерения метрической системы, в то время как другая — дюймовой. При написании приложений для поставки на международный рынок следует учитывать различия культур и регионов.

Разные культуры имеют разные календари и используют различные форматы чисел и дат. К тому же сортировка строк может давать разные результаты, потому что порядок A–Z может варьироваться в зависимости от культуры. Чтобы приложения отвечали требованиям глобального рынка, их следует глобализировать и локализировать.

*Глобализация* касается интернационализации приложений, то есть подготовки их для международных рынков. Глобализованные приложения поддерживают множество форматов чисел и дат, в зависимости от культуры, различные календари и тому подобное. *Локализация* — это адаптация приложения к требованиям конкретных культур. Для перевода строк можно использовать ресурсы.

.NET поддерживает глобализацию и локализацию как Windows, так и Web-приложений. Чтобы глобализовать приложение, можно использовать классы из пространства имен `System.Globalization`; чтобы локализовать приложение, можно использовать ресурсы, которые поддерживает пространство имен `System.Resources`.

Эта глава посвящена глобализации и локализации приложений .NET. Точнее говоря, здесь обсуждаются следующие вопросы:

- Применение классов, представляющих культуры и регионы.
- Интернационализация приложений.
- Локализация приложений.

# Пространство имен `System.Globalization`

Пространство имен `System.Globalization` включает все классы культур и регионов, поддерживающие различные форматы дат, чисел и даже различные календари, которые представлены такими классами, как `GregorianCalendar`, `HebrewCalendar`, `JapaneseCalendar` и так далее. Используя эти классы, можно получить различные отображения информации в зависимости от локальных настроек пользователя.

В этом разделе мы рассмотрим следующие понятия и соглашения, используемые пространством имен `System.Globalization`:

- Использование Unicode.
- Культуры и регионы.
- Пример, демонстрирующий все культуры и их характеристики.
- Сортировка.

## Использование Unicode

Символы Unicode занимают 16 бит, поэтому в этой кодировке есть место для 65 536 символов. Достаточно ли этого для всех языков, используемых в информационных технологиях? В случае китайского языка, например, необходимо более 80 000 символов. Однако Unicode спроектирована так, что справляется с этой задачей. Unicode различает базовые символы и комбинированные символы. Можно добавить множество комбинированных символов к одному базовому, чтобы построить отдельный отображаемый символ или текстовый элемент.

Возьмем, к примеру, исландский символ Ogonek. Ogonek может быть скомбинирован за счет использования базового символа 0x006F (латинская маленькая буква “o”) в сочетании с 0x0328 (комбинированный Ogonek) и 0x0304 (комбинированный знак долготы над гласными — Macron), как показано на рис. 17.1. Комбинированные символы определены в диапазоне от 0x0300 до 0x0345. Для рынков Америки и Европы существуют предопределенные символы, облегчающие работу со специальными символами. Символ Ogonek также задан предопределенным символом 0x01ED.

Для азиатского рынка, где только для одного китайского языка необходимо более 80 000 символов, такие предопределенные символы не существуют. В случае азиатских языков всегда используются комбинированные символы. Проблема, вытекающая из этого, заключается в том, что трудно получить правильное число отображаемых символов или текстовых элементов, а также в получении базовых символов вместо комбинированных. Пространство имен `System.Globalization` предлагает класс `StringInfo`, который можно использовать, чтобы справиться с этим.

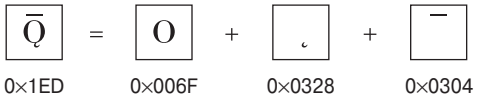


***Рис. 17.1. Исландский символ Ogonek***

В табл. 17.1 представлен список статических методов класса `StringInfo`, которые помогают работать с комбинированными символами.

**Таблица 17.1. Статические методы класса `StringInfo`**

| Метод | Описание |
|---|---|
| `GetNextTextElement` | Возвращает первый текстовый элемент (базовый символ и все комбинированные) указанной строки. |
| `GetTextElementEnumerator` | Возвращает объект `TextElementEnumerator`, позволяющий выполнить итерацию по всем текстовым элементам строки. |
| `ParseCombiningCharacters` | Возвращает массив целых чисел, ссылающихся на все базовые символы строки. |

Единственный отображаемый символ может содержать множество символов Unicode. Чтобы учесть это обстоятельство, когда вы пишете приложение для международного рынка, не используйте тип данных `char`. Используйте вместо него `string`, потому что он может содержать текстовый элемент, который включает как базовые символы, так и комбинированные, тогда как `char` — не может.

## Культуры и регионы

Мир разделен на множество культур и регионов, и приложения должны учитывать их различия. Культурой мы называем набор предпочтений, основанный на языке и культурных традициях. Документ RFC 1766 определяет наименования культур, используемые во всем мире, в зависимости от языка и страны или региона. Некоторые примеры: en-AU, en-CA, en-GB и en-US для английского языка в Австралии, Канаде, Великобритании и Соединенных Штатах соответственно.

Возможно, самым важным классом в пространстве имен `System.Globalization` является `CultureInfo`. Класс `CultureInfo` представляет культуру и определяет календари, форматирование дат и чисел, а также порядок сортировки строк, используемые в культуре.

Класс `RegionInfo` представляет региональные настройки (подобные валюте), а также указывает, применяется ли в регионе метрическая система. В некоторых регионах используется несколько языков. Одним из примеров может быть Испания, в которой есть баскская (eu-ES), каталонская (ca-ES), испанская (es-ES) и галисийская (gl-ES) культуры. Подобно тому, как один регион может иметь множество языков, также и на одном языке могут говорить во многих регионах, например, на испанском говорят в Мексике, Испании, Гватемале, Аргентине, Перу, и не только.

Далее в этой главе мы продемонстрируем пример приложения, демонстрирующего эти характеристики культур и регионов.

### Специфические, нейтральные и инвариантные культуры

Работая с .NET Framework, следует различать три типа культур: *специфические*, *нейтральные* и *инвариантные*.

Специфическая культура ассоциирована с реально существующей культурой, определенной в RFC 1766, как мы видели это в предыдущем разделе. Специфическая культура может быть отображена на нейтральную культуру. Например, de — это нейтральная культура для специфических культур de-AT, de-DE, de-CH и других. Здесь 'de' — сокращение, символизирующее немецкий язык, а AT, DE и CH — сокращения для Австрии, Германии и Швейцарии.

При переводе приложений, как правило, нет необходимости выполнять перевод для каждого региона; нет большой разницы между вариантами немецкого языка, принятыми в Германии и Австрии. Вместо применения специфических культур для локализации приложений достаточно использовать нейтральные культуры.

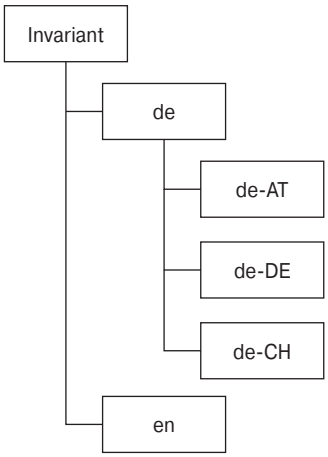


***Рис. 17.2. Отношения между типами культур***

Инвариантная культура независима от реальной культуры. Сохранение форматированных чисел и дат в файлах или отправка их по сети на сервер, используя культуру, независимую от любых пользовательских настроек — наилучший выбор.

На рис. 17.2 показаны отношения между типами культур.

### `CurrentCulture` и `CurrentUICulture`

При установке культуры необходимо различать культуру для пользовательского интерфейса и культуру для формата чисел и дат. Культуры ассоциированы с потоками, и с этими двумя типами культур к потоку можно применять по две настройки культур. Класс `Thread` имеет свойство `CurrentCulture` и `CurrentUICulture`. Свойство `CurrentCulture` предназначено для установки текущей культуры, используемой для форматирования и сортировки, в то время как `CurrentUICulture` служит для установки языка пользовательского интерфейса.

Пользователи могут изменять установки по умолчанию `CurrentCulture`, используя региональные и языковые настройки в панели управления Windows (рис. 17.3). С помощью этой конфигурации также можно изменять значения форматов по умолчанию, применяемых для вывода чисел, времени и дат в данной культуре.

`CurrentUICulture` не зависит от этих настроек. Установки `CurrentUICulture` зависят от языка операционной системы. Однако есть одно исключение: если с Windows XP или Windows 2000 установлен многоязыковый пользовательский интерфейс (MUI), то можно изменять язык пользовательского интерфейса настройками региональной конфигурации, и это оказывает влияние на свойство `CurrentUICulture`.

Все эти настройки определяют очень хорошие значения по умолчанию, и в большинстве случаев нет необходимости изменять их поведение. Если культура должна быть изменена, это легко сделать, изменив оба свойства потока, связанные с культурой, скажем, на испанскую культуру, как показано в следующем фрагменте кода:

```
System.Globalization.CultureInfo ci = new
  System.Globalization.CultureInfo("es-ES");
System.Threading.Thread.CurrentThread.CurrentCulture = ci;
System.Threading.Thread.CurrentThread.CurrentUICulture = ci;
```

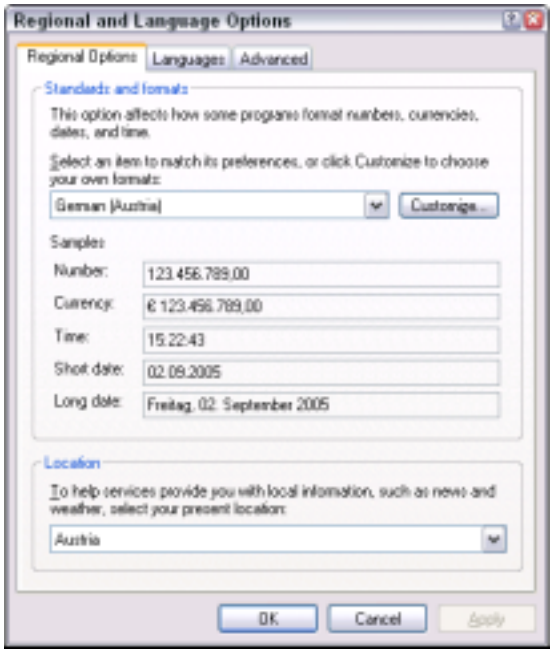


*Рис. 17.3. Региональные и языковые настройки Windows*

Теперь, когда мы узнали о настройках культуры, в следующем разделе поговорим о форматировании чисел и дат, на которое влияют установки `CurrentCulture`.

## Форматирование чисел

Числовые структуры `Int16`, `Int32`, `Int64` и тому подобные в пространстве имен `System` включают перегруженный метод `ToString()`. Этот метод может быть использован для создания различных представлений чисел, в зависимости от локальной настройки. Для структуры `Int32` `ToString()` перегружен следующими четырьмя версиями:

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

`ToString()` без аргументов возвращает строку без опций форматирования. Можно также передать строку и класс, реализующий интерфейс `IFormatProvider`.

Строка специфицирует формат представления. Формат может быть задан стандартной строкой форматирования чисел или же шаблонной строкой форматирования чисел. Для стандартной строки форматирования строки предопределены — в них `C` указывает нотацию валюты, `D` — число в десятичном формате, `E` — научную нотацию, `F` — формат с фиксированной точкой, `G` — общий вывод, `N` — вывод числа, `X` — шестнадцатеричный вывод. Шаблонное форматирование вывода чисел позволяет специфицировать количество разрядов, разделители групп и дробной части, процентную нотацию и так далее. Строка шаблонного формата `###.###` означает два трехзначных десятичных блока, разделенных групповым разделителем.

Интерфейс IFormatProvider реализован классами NumberFormatInfo, DateTimeFormatInfo и CultureInfo. Этот интерфейс определяет единственный метод — GetFormat(), возвращающий объект формата.

NumberFormatInfo может быть использован для определения специализированных заказных форматов чисел. Конструктор по умолчанию NumberFormatInfo создает культурно-независимый или инвариантный объект.

Используя NumberFormatInfo, можно изменить все опции форматирования — такие как знак положительных чисел, символ процента, разделитель групп числа, символ валюты и многое другое. Доступный только для чтения, культурно-независимый объект NumberFormatInfo возвращается статическим свойством InvariantInfo.

Объект NumberFormatInfo, в котором значения форматирования основаны на свойстве CultureInfo текущего потока, возвращается статическим свойством CurrentInfo.

Чтобы создать следующий пример, начнем с простого консольного проекта. В этом коде первый пример выводит число в формате культуры текущего потока (здесь — English-US, установка операционной системы). Второй пример использует метод ToString() с аргументом IFormatProvider. CultureInfo реализует IFormatProvider, поэтому создадим объект CultureInfo, используя культуру French. Третий пример изменяет культуру текущего потока. Культура изменяется на German с помощью свойства CurrentCulture экземпляра Thread:

```
using System;
using System.Globalization;
using System.Threading;
namespace Wrox.ProCSharp.Localization
{
   class Program
   {
     static void Main(string[] args)
     {
          int val = 1234567890;
          // культура текущего потока
          Console.WriteLine(val.ToString("N"));
          // использовать IFormatProvider
          Console.WriteLine(val.ToString("N",
                                new CultureInfo("fr-FR")));
          // изменить культуру потока
          Thread.CurrentThread.CurrentCulture =
                                new CultureInfo("de-DE");
          Console.WriteLine(val.ToString("N"));
     }
   }
}
```

Вывод показан на рис. 17.4. Вы можете сравнить форматы вывода для установок американского английского, французского и немецкого.

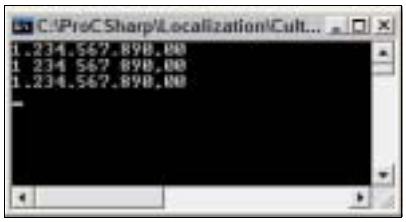


*Рис 17.4. Результат изменения культуры потока*

## Форматирование дат

Такая же поддержка, как для чисел, предусмотрена и для дат. Структура DateTime включает некоторые методы для преобразования

дат в строки. Общедоступные нестатические методы `ToLongDateString()`, `ToLongTimeString()`, `ToShortDateString()` и `ToShortTimeString()` создают строковые представления, используя текущую культуру. Метод `ToString()` можно применять для назначения другой культуры:

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

В строчном аргументе метода `ToString()` можно специфицировать предопределенный символ формата или заказную строку формата для преобразования даты в строку. Класс `DateTimeFormatInfo` специфицирует возможные значения. Аргументом `IFormatProvider` можно указать культуру. Применение перегрузок этого метода без аргумента `IFormatProvider` предполагает использование культуры текущего потока:

```
DateTime d = new DateTime(2005, 08, 09);

// текущая культура
Console.WriteLine(d.ToLongDateString());

// использовать IFormatProvider
Console.WriteLine(d.ToString("D", new CultureInfo("fr-FR")));

// использовать культуру текущего потока
CultureInfo ci = Thread.CurrentThread.CurrentCulture;
Console.WriteLine(ci.ToString() + ": " + d.ToString("D"));

ci = new CultureInfo("es-ES");
Thread.CurrentThread.CurrentCulture = ci;
Console.WriteLine(ci.ToString() + ": " + d.ToString("D"));
```

Вывод этого примера показывает методом `ToLongDateString()` текущую культуру потока, затем французскую версию, где экземпляр `CultureInfo` передается методу `ToString()`, и испанскую версию, где свойство `CurrentCulture` текущего потока изменяется на es-ES (рис. 17.5).

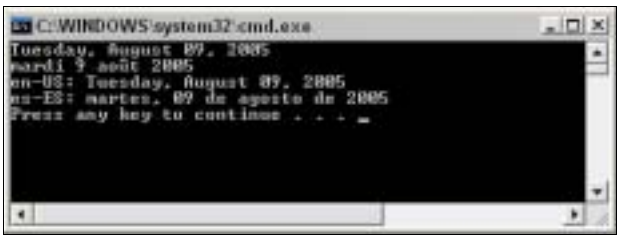


***Рис. 17.5. Текущая культура потока и ее изменение на французскую и испанскую версии***

## Культуры в действии

Чтобы увидеть все культуры в действии, используем пример приложения Windows Forms, которое перечисляет все культуры и демонстрирует разные характеристики свойство культуры. На рис. 17.6 показан пользовательский интерфейс приложения в среде Visual Studio 2005 Forms Designer.

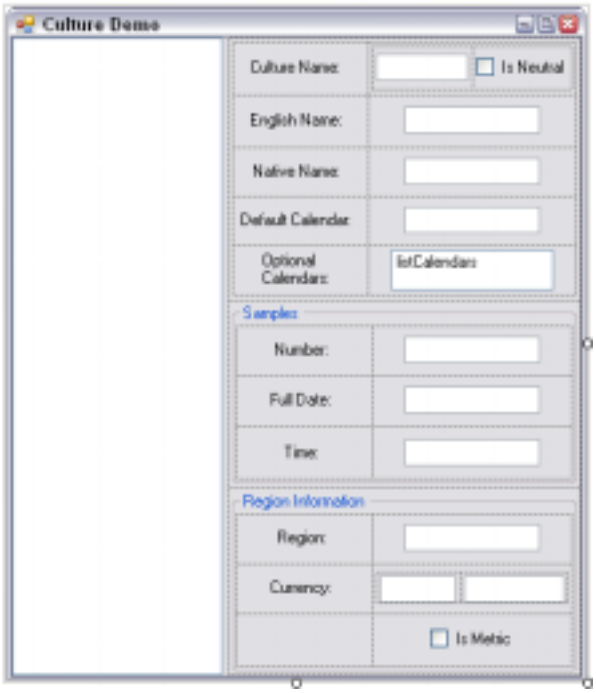


*Рис. 17.6. Пользовательский интерфейс примера приложения*

В процессе инициализации приложения все доступные культуры добавляются в древовидное представление, которое помещается в левой части окна приложения. Эта инициализация происходит в методе `AddCulturesToTree()`, который вызывается в конструкторе класса формы `CultureDemoForm`:

```
public CultureDemoForm()
{
  InitializeComponent();
  AddCulturesToTree();
}
```

В методе `AddCulturesToTree()` список всех культур получается от статического метода `CultureInfo.GetCultures()`.

Передавая этому методу аргумент `CultureTypes.AllCultures`, можно получить массив всех доступных культур. В цикле `foreach` каждая отдельная культура добавляется в представление дерева. Объект `TreeNode` создается для каждой отдельной культуры, потому что класс `TreeView` использует `TreeNode` для отображения узлов. Свойство `Tag` объекта `TreeNode` устанавливается в значение объекта `CultureInfo`, чтобы можно было позднее получить объект `CultureInfo` изнутри дерева.

Куда именно в дерево добавляется `TreeNode` — зависит от типа культуры. Если это нейтральная или инвариантная культура, то она добавляется к корневым узлам дерева. `TreeNode`, представляющие специфические культуры, добавляются к узлам их родительских нейтральных культур:

```
// внести все культуры в древовидное представление
public void AddCulturesToTree()
{
  // получить все культуры
  CultureInfo[] cultures =
     CultureInfo.GetCultures(CultureTypes.AllCultures);
  Array.Sort(cultures, new CultureComparer());
  TreeNode[] nodes = new TreeNode[cultures.Length];
  int i = 0;
  TreeNode parent = null;
  foreach (CultureInfo ci in cultures)
  {
     nodes[i] = new TreeNode();
     nodes[i].Text = ci.DisplayName;
     nodes[i].Tag = ci;
     if (ci.IsNeutralCulture)
     {
       // запомнить нейтральные культуры как родителей
       // для последующих
       parent = nodes[i];
       treeCultures.Nodes.Add(nodes[i]);
     }
     else if (ci.ThreeLetterISOLanguageName ==
       CultureInfo.InvariantCulture.ThreeLetterISOLanguageName)
     {
       // инвариантные культуры не имеют родителей
       treeCultures.Nodes.Add(nodes[i]);
     }
     else
     {
       // специфические культуры добавляются к узлам родительских нейтральных
       parent.Nodes.Add(nodes[i]);
     }
     i++;
  }
}
```

Когда пользователь выбирает узел в дереве, вызывается обработчик события `AfterSelect` элемента управления `TreeView`. Здесь обработчик реализован в методе `OnSelectCulture()`. Внутри этого метода все поля формы очищаются вызовом `ClearTextFields()`, прежде чем получить из дерева объект `CultureInfo` выбором свойства `Tag` узла `TreeNode`. Затем устанавливается значения свойств `Name`, `NativeName` и `EnglishName` объекта `CultureInfo` в текстовые поля формы. Если `CultureInfo` описывает нейтральную культуру, что можно проверить, прочитав значение свойства `IsNeutralCulture`, устанавливается флажок:

```
private void OnSelectCulture(object sender,
           System.Windows.Forms.TreeViewEventArgs e)
{
    ClearTextFields();
    // получить из дерева объект CultureInfo
    CultureInfo ci = (CultureInfo)e.Node.Tag;
    textName.Text = ci.Name;
    textNativeName.Text = ci.NativeName;
    textEnglishName.Text = ci.EnglishName;
    checkIsNeutral.Checked = ci.IsNeutralCulture;
```

Затем получаем информацию о календаре, принятом в данной культуре. Свойство `Calendar` класса `CultureInfo` возвращает объект `Calendar` по умолчанию для данной культуры. Поскольку класс `Calendar` не имеет свойства, сообщающего его имя, используем метод `ToString()` базового класса для получения имени класса, исключаем из него наименование пространства имен и помещаем в текстовое поле `textCalendar`.

Поскольку одна и та же культура может поддерживать множество календарей, свойство `OptionalCalendars` возвращает массив дополнительных поддерживаемых объектов `Calendar`. Эти необязательные календари отображаются в окне списка `listCalendars`. Класс `GregorianCalendar`, унаследованный от `Calendar`, имеет дополнительное свойство `CalendarType`, которое указывает тип григорианского календаря. Этот тип принимает значения из перечисления `GregorianCalendarTypes`: `Arabic`, `MiddleEastFrench`, `TransliteratedFrench`, `USEnglish` или `Localized`, в зависимости от культуры. Для григорианских календарей их тип также отображается в этом окне списка:

```
// календарь по умолчанию
textCalendar.Text = ci.Calendar.ToString().Remove(0, 21);
// наполнить список необязательных календарей
listCalendars.Items.Clear();
foreach (Calendar optCal in ci.OptionalCalendars)
{
  string calName = optCal.ToString().Remove(0, 21);
  // для GregorianCalendar добавить информацию о типе
  if (optCal is System.Globalization.GregorianCalendar)
  {
       GregorianCalendar gregCal = optCal as GregorianCalendar;
       calName += " " + gregCal.CalendarType.ToString();
  }
  listCalendars.Items.Add(calName);
}
```

Далее мы проверяем, является ли данная культура специфической (не нейтральной), используя условие `!ci.IsNeutralCulture` в операторе `if`.

Метод `ShowSamples()` отображает примеры форматирования числа и дат. Этот метод реализован в следующем разделе кода. Метод `ShowRegionInformation()` отображает некоторую информацию о регионе. Для инвариантной культуры можно отобразить только примеры форматирования чисел и дат, но не информацию о регионе. Инвариантная культура не связана ни с каким реальным языком, а потому не ассоциируется ни с каким регионом:

```
// отобразить примеры чисел и дат
if (!ci.IsNeutralCulture)
{
     groupSamples.Enabled = true;
     ShowSamples(ci);
     // инвариантная культура не имеет региона
     if (ci.ThreeLetterISOLanguageName == "IVL")
     {
          groupRegionInformation.Enabled = false;
     }
     else
     {
```

```
            groupRegionInformation.Enabled = true;
            ShowRegionInformation(ci.LCID);
        }
    }
    else // нейтральная культура не имеет ни региона,
         // ни форматирования чисел/дат
    {
        groupSamples.Enabled = false;
        groupRegionInformation.Enabled = false;
    }
}
```

Чтобы показать локализованные примеры чисел и дат, выбранный объект типа `CultureInfo` передается методу `ToString()` в аргументе `IFormatProvider`:

```
private void ShowSamples(CultureInfo ci)
{
  double number = 9876543.21;
  textSampleNumber.Text = number.ToString("N", ci);
  DateTime today = DateTime.Today;
  textSampleDate.Text = today.ToString("D", ci);
  DateTime now = DateTime.Now;
  textSampleTime.Text = now.ToString("T", ci);
}
```

Чтобы отобразить информацию, ассоциированную с объектом `RegionInfo`, в методе `ShowRegionInformation()` объект `RegionInfo` конструируется, принимая идентификатор выбранной культуры. Затем для отображения информации обращаемся к свойствам `DisplayName`, `CurrencySymbol`, `ISOCurrencySymbol` и `IsMetric`:

```
private void ShowRegionInformation(int culture)
{
  RegionInfo ri = new RegionInfo(culture);
  textRegionName.Text = ri.DisplayName;
  textCurrency.Text = ri.CurrencySymbol;
  textCurrencyName.Text = ri.ISOCurrencySymbol;
  checkIsMetric.Checked = ri.IsMetric;
}
```

Запустив это приложение на выполнение, мы увидим все доступные культуры в древовидном представлении, и, выбрав любую из них, сможем просмотреть все ее характеристики, как показано на рис. 17.7.

## Сортировка

Порядок сортировки строк зависит от культуры. Некоторые культуры имеют отличающиеся порядки сортировки. Одним примером может служить финский язык, в котором V и W трактуются, как одно и то же. Алгоритмы, сравнивающие строки с целью сортировки, по умолчанию используют сортировку, зависящую от культуры.

Чтобы продемонстрировать поведение сортировки для финского языка, следующий код представляет маленький пример консольного приложения, в котором некоторые наименования штатов США помещаются в массив в произвольном порядке. Мы будем использовать классы из пространств имен `System.Collections`, `System.Threading` и `System.Globalization`, поэтому они должны быть доступны.

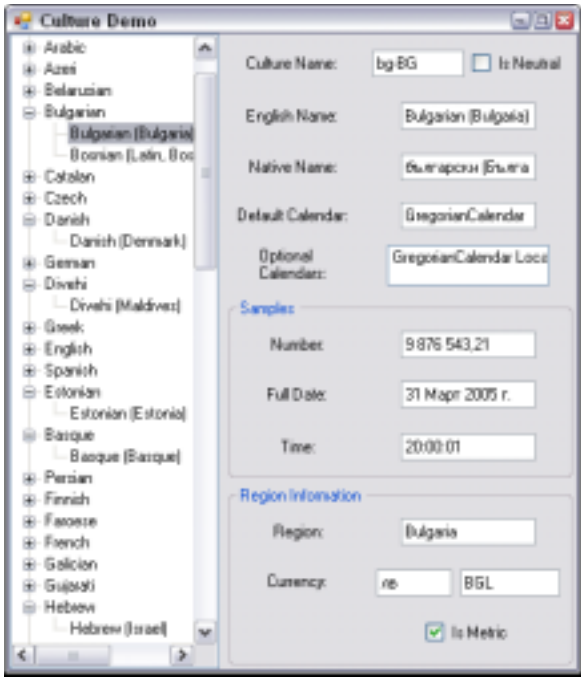


*Рис. 17.7. Результат выполнение примера приложения*

Приведенный ниже метод `DisplayNames()` применяется для отображения элементов массива или коллекции на экране:

```
static void DisplayNames(IEnumerable e)
{
  foreach (string s in e)
       Console.Write(s + " - ");
  Console.WriteLine();
}
```

В методе `Main()` после создания массива с наименованиями некоторых штатов США свойству потока `CurrentCulture` присваивается культура Finnish, так что последующий вызов `Array.Sort()` использует финский порядок сортировки строк. Вызов метода `DisplayNames()` отобразит все наименования штатов на консоли:

```
static void Main(string[] args)
{
  string[] names = {"Alabama", "Texas", "Washington",
                    "Virginia", "Wisconsin", "Wyoming",
                    "Kentucky", "Missouri", "Utah", "Hawaii",
                    "Kansas", "Lousiana", "Alaska", "Arizona"};
  Thread.CurrentThread.CurrentCulture =
       new CultureInfo("fi-FI");
  Array.Sort(names);
  Console.WriteLine("\nотсортировано...");
  DisplayNames(names);
```

После первого отображения наименований штатов в порядке финской сортировки массив сортируется снова. Если нужно получить сортировку, независимую от культуры пользователя, что может быть удобно, когда сортированный массив отправляется на сервер или сохраняется где-то еще, то в этом случае можно использовать инвариантную культуру.

Это можно сделать, передав второй аргумент `Array.Sort()`. Метод `Sort()` принимает во втором аргументе объект, реализующий интерфейс `IComparer`. Класс `Comparer` из пространства имен `System.Collections` реализует `IComparer`. `Comparer.DefaultInvariant` возвращает объект `Comparer`, использующий инвариантную культуру для сравнения значений элементов массива для независимой от культуры сортировки:

```
            // сортировка с использованием инвариантной культуры
            Array.Sort(names, Comparer.DefaultInvariant);
            Console.WriteLine("\nотсортировано для инвариантной культуры...");
            DisplayNames(names);
        }
```

На рис. 17.8 показан вывод этой программы: здесь можно видеть результат финской сортировки и затем — сортировки, независимой от культуры. Легко заметить, что в первом случае Washington идет перед Virginia.

***Рис. 17.8. Результат нескольких видов сортировки***

> Если сортировка коллекции должна быть независимой от культуры, ее следует сортировать в инвариантной культуре. В частности, это может быть удобно, когда отсортированный результат отправляется на сервер либо сохраняется в файле.

В дополнение к форматированию и системе единиц измерения, принятых в той или иной местности, также в зависимости от культуры могут различаться тексты и изображения, используемые программой. И здесь в игру вступают ресурсы.

# Ресурсы

Ресурсы — такие как изображения и таблицы строк — могут быть помещены в ресурсные файлы или подчиненные сборки. Эти ресурсы могут быть очень полезны при локализации приложений, и .NET имеет встроенную поддержку для поиска локализованных ресурсов.

Прежде чем мы увидим, как использовать ресурсы для локализации приложений, в следующем разделе поговорим о том, как создаются и читаются ресурсы, не принимая во внимание аспекты языка.

## Создание ресурсных файлов

Ресурсные файлы могут содержать такие вещи, как изображения и таблицы строк. Ресурсный файл создается либо как обычный тестовый файл, либо как файл с расширением `.resX`, использующий XML. Начнем этот раздел с примера текстового файла.

Ресурс, хранящий таблицу строк, может быть создан как обычный текстовый файл. В нем строкам просто назначаются ключи. Ключ — это имя, которое может быть использовано программой для получения значения. Как в ключах, так и в значениях допускаются пробелы.

Следующий пример показывает простую таблицу строк в файле `strings.txt`:

```
Title = Professional C#
Chapter = Localization
Author = Christian Nagel
Publisher = Wrox Press
```

## Генератор ресурсных файлов

Генератор ресурсных файлов (утилита `Resgen.exe`) используется для создания ресурсного файла из `strings.txt`. Следующая команда:

```
resgen strings.txt
```

создает `strings.resources`. Результирующий ресурсный файл может быть добавлен к сборке — либо как внешний файл, либо как встроенный в DLL или EXE. Утилита `Resgen` также поддерживает создание базирующихся на XML ресурсных файлов `.resX`. Простейший способ создания XML-файла:

```
resgen strings.txt strings.resX
```

Эта команда создает ресурсный файл XML `strings.resX`. Позднее в этой главе, в разделе "Пример локализации с применением Visual Studio" будет показано, как работать с ресурсными файлами XML.

В .NET 2.0 утилита Resgen поддерживает строго типизированные ресурсы. Строго типизированный ресурс представляется классом, который обращается к ресурсам. Этот класс может быть создан автоматически, если применить опцию `/str` к утилите `Resgen`:

```
resgen /str:C#,DemoNamespace,DemoResource,DemoResource.cs strings.resX
```

Вместе с опцией `/str` указываются язык, пространство имен, имя класса и имя файла исходного кода.

Утилита `Resgen` не поддерживает добавление изображений. Среди примеров SDK .NET Framework есть пример `ResGen`, включающий руководство. С помощью `ResGen` можно включить ссылки на изображения в файл `.resX`. Добавление изображений также может быть выполнено программно, если использовать классы `ResourceWriter` или `ResXResourceWriter`, как мы увидим это позже.

### ResourceWriter

Вместо использования утилиты Resgen для построения ресурсных файлов, можно написать простенькую программу. Класс `ResourceWriter` из пространства имен `System.Resources` применяется для записи двоичных ресурсных файлов;

ResXResourceWriter пишет XML-ориентированные ресурсные файлы. Оба эти класса поддерживают изображения и любые другие сериализируемые объекты. Для использования ResXResourceWriter потребуется включить ссылку на сборку System.Windows.Forms.

В следующем примере кода мы создадим объект ResXResourceWriter с именем rw, используя конструктор с именем файла Demo.resx. После создания экземпляра ему можно добавить множество ресурсов общим объемом до 2 Гбайт, используя для этого метод AddResource() класса ResXResourceWriter. Первый аргумент AddResource() указывает имя ресурса, а второй — значение. Ресурс изображения может быть добавлен посредством экземпляра класса Image. Чтобы использовать класс Image, необходимо сослаться на сборку System.Drawing. Также необходимо использовать директиву using, чтобы открыть пространство имен System.Drawing.

Создадим объект Image, открыв файл logo.gif. Нужно либо скопировать изображение в каталог исполняемой программы, либо указывать полное путевое имя файла изображения в аргументе метода Image.ToFile(). Оператор using указывает, что ресурс изображения должен автоматически освобождаться в конце блока using. Дополнительные простые строковые ресурсы добавляются к объекту ResXResourceWriter. Метод Close() класса ResXResourceWriter автоматически вызывает ResXResourceWriter.Generate(), чтобы записать ресурсы в файл Demo.resx:

```
using System;
using System.Resources;
using System.Drawing;
class Program
{
     static void Main()
     {
       ResXResourceWriter rw = new ResXResourceWriter("Demo.resx");
       using (Image image = Image.FromFile("logo.gif"))
       {
            rw.AddResource("WroxLogo", image);
            rw.AddResource("Title", "Professional C#");
            rw.AddResource("Chapter", "Localization");
            rw.AddResource("Author", "Christian Nagel");
            rw.AddResource("Publisher", "Wrox Press");
            rw.Close();
       }
     }
}
```

Запуск этой маленькой программы создает ресурсный файл Demo.resx, который включает в себя logo.gif. Эти ресурсы мы используем в следующем простом примере приложения Windows.

## Использование ресурсных файлов

Ресурсные файлы можно добавлять в сборки компилятором командной строки C# csc.exe, используя для этого опцию /resource, либо непосредственно в среде Visual Studio 2005. Чтобы посмотреть, как пользоваться ресурсными файлами в Visual Studio 2005, создадим приложение Windows на C# и назовем его ResourceDemo.

Для добавления в проект ранее созданного ресурсного файла `Demo.resources` служит контекстное меню проводника Solution Explorer (Add⇨Add Existing Item (Добавить⇨Добавить существующий элемент)). По умолчанию свойство `BuildAction` для этого ресурса установлено в `Embedded Resource`, поэтому он будет встроен в выходную сборку (рис. 17.9).

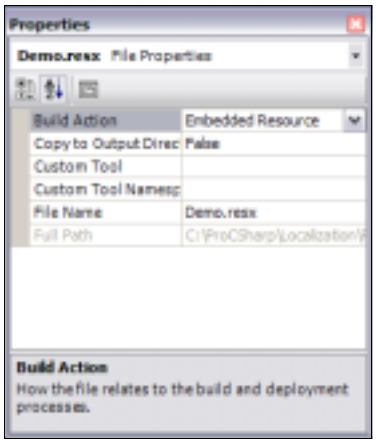


*Рис. 17.9. Свойство `BuildAction`*

После компоновки проекта можно проверить сгенерированную сборку утилитой `ildasm`, чтобы увидеть атрибут `.mresource` в манифесте (рис. 17.10). `.mresource` объявляет имя ресурса в сборке. Если `.mresource` объявлен как `public` (в данном примере это так), то ресурс экспортируется сборкой и может быть использован классами других сборок. Если `.mresource` объявлен как `private`, это значит, что он не экспортируется и доступен только в пределах данной сборки.

Когда ресурсы добавляются в сборку из Visual Studio 2005, они всегда объявлены как `public` (см. рис. 17.10). Если для создания сборок используется инструмент генерации сборок, то в этом случае можно с помощью опций командной строки выбрать уровень доступа к ресурсам — `public` или `private`.

Опция `/embed:demo.resources,Y` добавляет ресурс `public`, в то время как `/embed:demo.resources,N` — ресурс `private`.

Если сборка сгенерирована в Visual Studio 2005, то видимость ресурса можно изменить позже. Для этого воспользуйтесь `ilasm`, выберите команду меню File⇨Dump (Файл⇨Дамп) для открытия сборки и сгенерируйте исходный файл MSIL. Код MSIL можно изменить в простом текстовом редакторе. Подобным образом можно заменить `.mresource public` на `.mresource private`. Затем с помощью `ilasm` можно повторно сгенерировать сборку из исходного кода MSIL:

```
ilasm /exe ResourceDemo.il
```

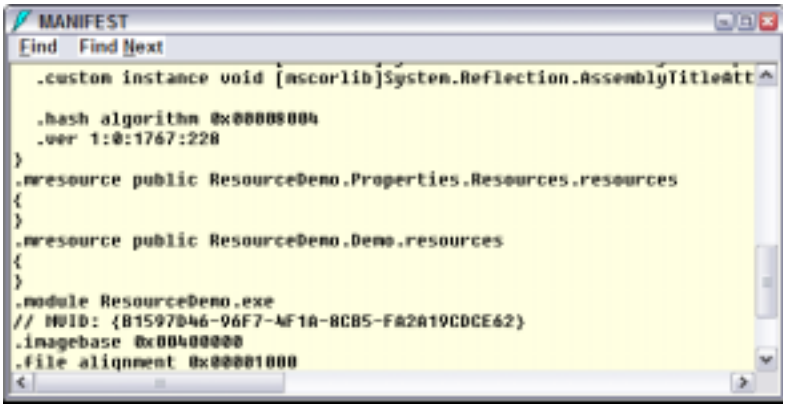


*Рис. 17.10. Манифест ресурса*

Добавим в Windows-приложении несколько текстовых полей и изображение, перетащив элементы Windows Forms из панели инструментов в конструктор. Значения ресурсов будут отображены в этих элементах Windows Forms. Изменим свойства `Text` и `Name` текстовых полей, а также их метки так, как показано в следующем коде. Свойство имени элемента управления `PictureBox` заменим на логотип. На рис. 17.11 показан окончательный вид формы в конструкторе Forms Designer. Элемент управления `PictureBox` показан в виде прямоугольника без сетки в верхнем левом углу.

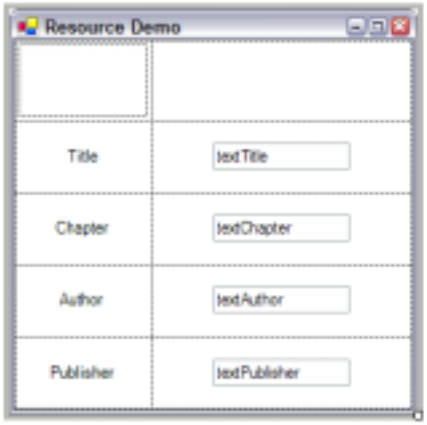


*Рис. 17.11. Окончательный вид формы*

Для доступа к встроенному ресурсу воспользуемся классом `ResourceManager` из пространства имен `System.Resources`. Конструктору класса `ResourceManager` в качестве аргумента можно передать сборку, содержащую ресурсы. В данном примере ресурсы встроены в исполняемую сборку, поэтому в качестве второго аргумента передаем результат, возвращенный методом `Assembly.GetExecutingAssembly()`. Первый аргумент — корневое имя ресурсов. Корневое имя состоит из пространства имен, дополненного именем ресурсного файла, но без расширения. Как мы уже видели ранее, `ildasm` показывает имя.

Все, что нужно сделать — исключить расширение `.resources` из показанного имени. Можно также получить имя программно, используя для этого метод `GetManifestResourceNames()` класса `System.Reflection.Assembly`:

```
using System.Reflection;
using System.Resources;
//...
partial class ResourceDemoForm : Form
{
  private System.Resources.ResourceManager rm;
  public ResourceDemoForm()
  {
     InitializeComponent();
     Assembly assembly = Assembly.GetExecutingAssembly();
     rm = new ResourceManager("ResourceDemo.Demo", assembly);
```

Применяя экземпляр `rm` класса `ResourceManager`, можно получать все ресурсы, указывая ключ в методах `GetObject()` и `GetString()`:

```
     logo.Image = (Image)rm.GetObject("WroxLogo");
     textTitle.Text = rm.GetString("Title");
     textChapter.Text = rm.GetString("Chapter");
     textAuthor.Text = rm.GetString("Author");
     textPublisher.Text = rm.GetString("Publisher");
  }
```

Запустив этот код, мы увидим строковые ресурсы и ресурс-изображение (рис. 17.12).

Как уже упоминалось, .NET 2.0 оснащен новым средством — строго типизированными ресурсами. Если применить строго типизированные ресурсы, то приведенный выше код конструктора класса `ResourceDemoForm` может быть упрощен; в этом случае нет необходимости создавать экземпляр `ResourceManager` и обращаться к ресурсам, используя индексы. Вместо имен к ресурсам можно обращаться через свойства:

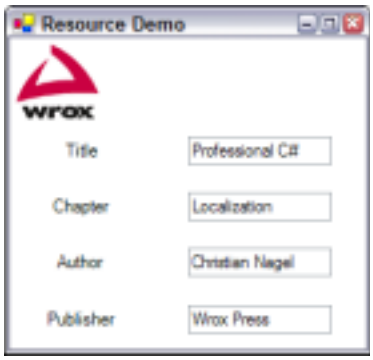


*Рис. 17.12. Результат просмотра ресурсов*

```
public ResourceDemoForm()
{
  InitializeComponent();
  pictureLogo.Image = Demo.WroxLogo;
  textTitle.Text = Demo.Title;
  textChapter.Text = Demo.Chapter;
  textAuthor.Text = Demo.Author;
  textPublisher.Text = Demo.Publisher;
}
```

Чтобы создать строго типизированный ресурс, свойство `Custom Tool` XML-ориентированного ресурсного файла должно быть установлено в `ResXFileCodeGenerator`. Установка этой опции генерирует класс `Demo` (он имеет то же имя, что и ресурс). Этот класс включает статические свойства для всех ресурсов, предоставляя, таким образом, для них строго типизированные имена. С реализацией статических свойств используется объект `ResourceManager`, который создается при первом доступе, а затем кэшируется:

```
/// <summary>
/// A strongly-typed resource class, for looking up localized strings, etc.
/// </summary>
// This class was auto-generated by the StronglyTypedResourceBuilder
// class via a tool like ResGen or Visual Studio.NET.
// To add or remove a member, edit your .ResX file then rerun ResGen
// with the /str option, or rebuild your VS project.
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
  "System.Resources.Tools.StronglyTypedResourceBuilder", "2.0.0.0")]
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
internal class Demo {
  private static global::System.Resources.ResourceManager resourceMan;
  private static global::System.Globalization.CultureInfo resourceCulture;
  [global::System.Diagnostics.CodeAnalysis.SuppressMessageAttribute(
    "Microsoft.Performance", "CA1811:AvoidUncalledPrivateCode")]
  internal Demo() {
  }
  /// <summary>
  /// Returns the cached ResourceManager instance used by this class.
  /// </summary>
  [global::System.ComponentModel.EditorBrowsableAttribute(
    global::System.ComponentModel.EditorBrowsableState.Advanced)]
  internal static global::System.Resources.ResourceManager
    ResourceManager {
    get {
      if ((resourceMan == null)) {
          global::System.Resources.ResourceManager temp =
```

```
            new global::System.Resources.ResourceManager(
            "ResourceDemo.Demo", typeof(Demo).Assembly);
            resourceMan = temp;
        }
        return resourceMan;
      }
    }
    /// <summary>
    /// Overrides the current thread's CurrentUICulture property for all
    /// resource lookups using this strongly typed resource class.
    /// </summary>
    [global::System.ComponentModel.EditorBrowsableAttribute(
      global::System.ComponentModel.EditorBrowsableState.Advanced)]
    internal static System.Globalization.CultureInfo Culture {
      get {
        return resourceCulture;
      }
      set {
        resourceCulture = value;
      }
    }
    /// <summary>
    /// Looks up a localized string similar to "Christian Nagel".
    /// </summary>
    internal static string Author {
      get {
        return ResourceManager.GetString("Author", resourceCulture);
      }
    }
    /// <summary>
    /// Looks up a localized string similar to "Localization".
    /// </summary>
    internal static string Chapter {
      get {
        return ResourceManager.GetString("Chapter", resourceCulture);
      }
    }
    /// <summary>
    /// Looks up a localized string similar to "Wrox Press".
    /// </summary>
    internal static string Publisher {
      get {
        return ResourceManager.GetString("Publisher", resourceCulture);
      }
    }
    /// <summary>
    /// Looks up a localized string similar to "Professional C#".
    /// </summary>
    internal static string Title {
      get {
        return ResourceManager.GetString("Title", resourceCulture);
      }
    }
    internal static System.Drawing.Bitmap WroxLogo {
      get {
        return ((System.Drawing.Bitmap)
          (ResourceManager.GetObject("WroxLogo", rescourceCulture)));
      }
    }
  }
```

## Пространство имен `System.Resources`

Прежде чем перейти к следующему примеру, в этом разделе мы завершим обзор классов, содержащихся в пространстве имен `System.Resources`, которые имеют дело с ресурсами:

- Класс `ResourceManager` может быть использован для получения ресурсов для текущей культуры из сборок или ресурсных файлов. Применяя `ResourceManager`, можно получить `ResourceSet` для определенной конкретной культуры.
- Класс `ResourceSet` представляет набор ресурсов для определенной культуры. Когда создается экземпляр `ResourceSet`, он проходит по классу, реализующему интерфейс `IResourceReader`, и сохраняет все ресурсы в `HashTable`.
- Интерфейс `IResourceReader` используется из `ResourceSet` для перечисления ресурсов. Класс `ResourceReader` реализует этот интерфейс.
- Класс `ResourceWriter` применяется для создания ресурсного файла. `ResourceWriter` реализует интерфейс `IresourceWriter`.
- `ResXResourceSet`, `ResXResourceReader` и `ResXResourceWriter` подобны `ResourceSet`, `ResourceReader` и `ResourceWriter`; однако они используются для создания ресурсного файла, основанного на XML, `.resX`, вместо двоичного файла. Внутри XML-файла можно применять `ResXFileRef` для создания ссылки на ресурс вместо его встраивания.

# Пример локализации с применением Visual Studio

В этом разделе мы создадим простое приложение Windows, на примере которого продемонстрируем применение Visual Studio для локализации. Это приложение не использует сложных форм Windows и не содержит никакой полезной функциональности, потому что его главное назначение — продемонстрировать локализацию. В автоматически сгенерированном исходном коде изменим название пространства имен на `Wrox.ProCSharp.Localization`, а имя класса — на `BookOfTheDayForm`. Название пространства имен следует изменить не только в исходном файле `BookOfTheDay — Form.cs`, но также в настройках проекта, чтобы все сгенерированные исходные файлы также получили это пространство имен. Пространство имен для всех создаваемых элементов можно изменить, выбрав `Common Properties` в меню Project⇨Properties (Проект⇨Свойства).

> *Приложения Windows Forms подробно рассматриваются в главах 23, 24 и 25.*

Чтобы показать последствия локализации, эта программа использует изображение, некоторый текст, дату и число. Изображение показывает флаг, который также локализован. На рис. 17.13 показана форма приложения, как она выглядит в конструкторе Windows Forms.

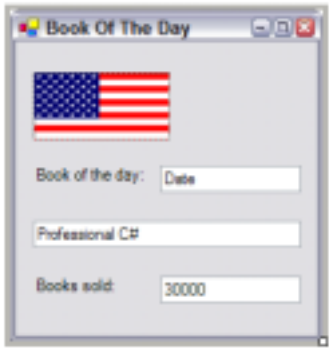


***Рис. 17.13. Форма в конструкторе Windows Forms***

В табл. 17.2 перечислены все значения свойств Name и Text элементов Windows Forms.

**Таблица 17.2. Значения свойств Name и Text элементов Windows Forms**

| Имя | Текст |
|---|---|
| LabelBookOfTheDay | Book of the day: (Книга дня:) |
| LabelItemsSold | Books sold: (Продано книг:) |
| TextDate | Date (Дата) |
| TextTitle | Professional C# |
| TextItemsSold | 30000 |
| PictureFlag | |

В дополнение к этой форме может понадобиться окно сообщения, которое отображает приглашающее сообщение. Это сообщение может изменяться в зависимости от текущего времени суток. Пример демонстрирует, что локализация динамически создаваемого диалога должна выполняться иначе. В методе WellcomeMessage() окно сообщений отображается с помощью MessageBox.Show(). Этот метод будет вызываться в конструкторе класса формы BookOfTheDayForm перед вызовом InitializeComponent().

Ниже представлен код метода WellcomeMessage():

```
public void WelcomeMessage()
{
  DateTime now = DateTime.Now;
  string message;
  if (now.Hour <= 12)
  {
     message = "Good Morning";
  }
  else if (now.Hour <= 19)
  {
     message = "Good Afternoon";
  }
  else
  {
     message = "Good Evening";
  }
  MessageBox.Show(message + "\nThis is a localization sample.");
}
```

Число и дата в форме должны быть установлены с применением опций форматирования. Для установки значений с опциями форматирования добавим метод SetDateAndNumber(). В реальном приложении эти значения могут быть получены от Web-службы или из базы данных, однако этот пример посвящен только локализации. Дата форматируется с использованием опции D (для отображения длинного названия даты). Число отображается с использованием строки-шаблона числового формата ###,###,###, где # представляет десятичную цифру, а запятая (,) является разделителем групп:

```
public void SetDateAndNumber()
{
  DateTime today = DateTime.Today;
  textDate.Text = today.ToString("D");
  int itemsSold = 327444;
  textItemsSold.Text = itemsSold.ToString("###,###,###");
}
```

В конструкторе класса `BookOfTheDayForm` вызываются оба метода — `WelcomeMessage()` и `SetDateAndNumber()`:

```
public BookOfTheDayForm()
{
  WelcomeMessage();
  InitializeComponent();
  SetDateAndNumber();
}
```

Магические свойства конструктора Windows Forms проявляются, когда мы устанавливаем свойство `Localizable` формы из `false` в `true`. При этом создается ресурсный XML-файл для диалогового окна, в который помещаются все ресурсные строки, свойства (включая расположение и размеры элементов Windows Forms), встроенные изображения и так далее.

Вдобавок к этому изменяется реализация метода `InitializeComponent()`; создается экземпляр класса `System.Resources.ResourceManager`, и для того, чтобы получить значения и позиции текстовых полей и изображений, используется метод `GetObject()` вместо записи значений непосредственно в код. Метод `GetObject()` обращается к свойству `CurrentUICulture` текущего потока, чтобы найти правильную локализацию ресурсов. Ниже представлена часть `InitializeComponent()` из файла `BookOfTheDayForm.Designer.cs`, предшествующая установке свойства `Localizable` в значение `true`, где устанавливаются все свойства `textboxTitle`:

```
private void InitializeComponent()
{
   //...
   this.textTitle = new System.Windows.Forms.TextBox();
   //...
   //
   // textTitle
   //
   this.textTitle.Location = new System.Drawing.Point(24, 152);
   this.textTitle.Name = "textTitle";
   this.textTitle.Size = new System.Drawing.Size(256, 20);
   this.textTitle.TabIndex = 2;
   this.textTitle.Text = "Professional C#";
```

После установки `Localizable` в значение `true` метод `IntializeComponent()` автоматически изменяется:

```
private void InitializeComponent()
{
   System.ComponentModel.ComponentResourceManager resources = new
     System.ComponentModel.ComponentResourceManager(
     typeof(BookOfTheDayForm));
   //...
   this.textTitle = new System.Windows.Forms.TextBox();
   //...
   resoures.ApplyResources(this.textTitle, "textTitle");
```

Откуда диспетчер ресурсов получит данные? Когда `Localizable` устанавливается в `true`, генерируется ресурсный файл `BookOfTheDay.resX`. В этом файле можно найти схему XML-ресурса, за которой следуют все элементы формы: `Type`, `Text`, `Location`, `TabIndex` и так далее.

Класс `ComponentResourceManager` наследуется от `ResourceManager` и представляет метод `ApplyResources()`. В `ApplyResources()` ресурсы, определенные во втором аргументе, применяются к объекту, переданному в первом аргументе.

Следующий сегмент XML показывает несколько свойств `textBoxTitle`: свойство `Location` имеет значение 13,133, свойство `TabIndex` имеет значение `2`, свойство `Text` установлено `Professional C#` и так далее. Вместе с каждым значением сохраняется его тип.

Например, свойство `Location` относится к типу `System.Drawing.Point`, и этот класс находится в сборке `System.Drawing`.

Почему местоположения и размеры сохраняются в файле XML? Дело в том, что при переводе многие строки полностью изменяются в размере, и не помещаются в исходных позициях. Когда местоположения и размеры сохраняются в ресурсном файле, то все, что необходимо для локализации, хранится вместе в этих файлах отдельно от кода C#:

```
<data name="textTitle.Anchor" type="System.Windows.Forms.AnchorStyles,
 System.Windows.Forms">
   <value>Bottom, Left, Right</value>
</data>
<data name="textTitle.Location" type="System.Drawing.Point, System.Drawing">
   <value>13, 133</value>
</data>
<data name="textTitle.Size" type="System.Drawing.Size, System.Drawing">
   <value>196, 20</value>
</data>
<data name="textTitle.TabIndex" type="System.Int32, mscorlib">
   <value>2</value>
</data>
<data name="textTitle.Text">
   <value xml:space="preserve">Professional C#</value>
</data>
```

Изменяя некоторые из этих ресурсных значений, нет необходимости работать напрямую с кодом XML. Эти ресурсы можно менять непосредственно в конструкторе Visual Studio 2005. Всякий раз, когда меняется свойство `Language` формы и свойства некоторых ее элементов, при этом генерируется новый ресурсный файл для указанного языка. Создадим немецкую версию формы, установив значение `German` для свойства `Language`, и французскую версию, установив для `Language` значение `French`. При этом для каждого языка получим ресурсный файл с измененными свойствами: `BookOfTheDayForm.de.resX` и `BookOfTheDayForm.fr.resX`.

В табл. 17.3 перечислены изменения, необходимые для получения немецкой версии.

В табл. 17.4 перечислены изменения, необходимые для получения французской версии.

**Таблица 17.3. Изменения свойств для получения немецкой версии**

| Имя German | Значение |
|---|---|
| `$this.Text (заголовок формы)` | `Buch des Tages` |
| `labelItemsSold.Text` | `Bücher verkauft:` |
| `labelBookOfTheDay.Text` | `Buch des Tages:` |

**Таблица 17.4. Изменения свойств для получения французской версии**

| Имя French | Значение |
|---|---|
| `$this.Text (заголовок формы)` | `Le livre du jour` |
| `labelItemsSold.Text` | `Des livres vendus:` |
| `labelBookOfTheDay.Text` | `Le livre du jour:` |

В .NET 2.0 теперь графические изображения по умолчанию не перемещаются в подчиненные сборки. Однако в нашем примере приложения изображение флага должно выбираться в соответствии со страной. Чтобы сделать это, необходимо добавить изображение американского флага в файл `Resources.resx`. Этот файл можно найти в разделе Properties (Свойства) проводника Visual Studio Solution Explorer. В редакторе ресурсов выберем категорию Images (Изображения), как показано на рис. 17.14, и добавим файл `americanflag.bmp`. Чтобы обеспечить возможность локализации, изображение должно иметь одно и то же имя для всех языков. У нас изображение в файле `Resources.resx` названо `Flag`. Изображение можно переименовать в редакторе свойств. В этом же редакторе также можно указать, должно ли изображение быть привязанным или встроенным. Для повышения производительности при работе с ресурсами изменено поведение Visual Studio 2005 по умолчанию — теперь по умолчанию изображения привязываются, а не встраиваются. При этом файл изображения должен поставляться вместе с приложением. Если вы хотите встроить изображения в сборку, то измените значение свойства `Persistence` на `Embedded`.

Локализованные версии флагов могут быть добавлены копированием `Resource.resx` в `Resource.de.resx` и `Resource.fr.resx` с заменой имен файлов изображений, соответственно, на `GermanFlag.bmp` и `FranceFlag.bmp`.

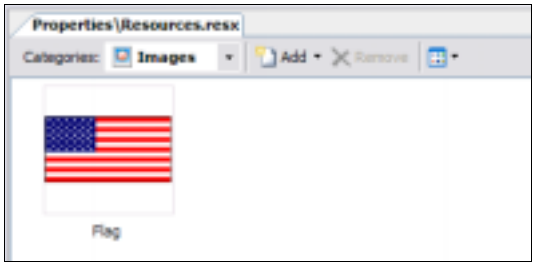


*Рис. 17.14. Выбор категории Images в редакторе ресурсов*

Поскольку строго типизированный ресурсный класс необходим только с нейтральным ресурсом, свойство `CustomTool` может быть очищено в ресурсных файлах специфических языков.

Теперь при компиляции проекта будет создана *подчиненная сборка* (satellite assembly) для каждого языка. Внутри отладочного каталога (или в каталоге выпуска, в зависимости от активной конфигурации) создаются подкаталоги с именами вроде `de` и `fr`. В этих подкаталогах вы найдете файл `BookOfTheDay.resources.dll`. Этот файл представляет собой подчиненную сборку, которая содержит в себе только локализованные ресурсы. Если открыть эту сборку утилитой `ildasm` (рис. 17.15), можно увидеть манифест со встроенными ресурсами и определенной локальной настройкой. Сборка имеет локальную настройку `de` в своих атрибутах, и потому может быть найдена в подкаталоге `de`. Также можно увидеть имя ресурса с `.mresource`; оно снабжено префиксом — именем пространства имен `Wrox.ProCSharp.Localization`, за которым следует имя класса `BookOfTheDayForm` и код языка `de`.

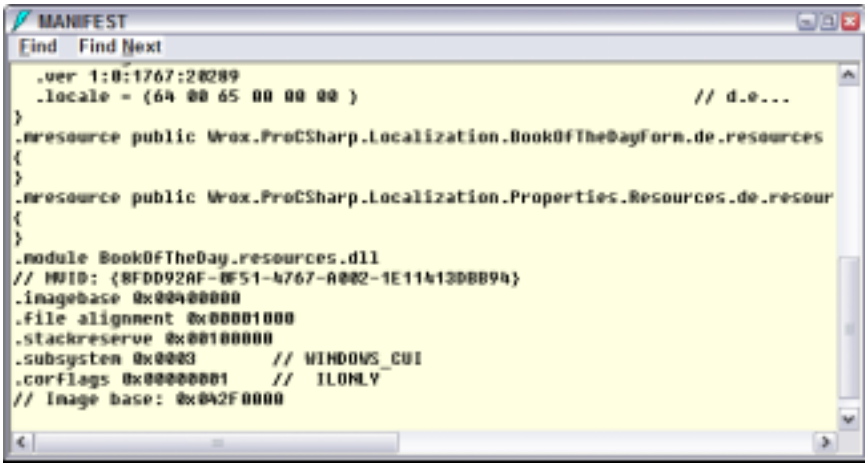


***Рис. 17.15. Манифест со встроенными ресурсами***

## Программное переключение культуры

После перевода ресурсов и построения подчиненных сборок, мы получаем корректный перевод в зависимости от настроек культуры на машине пользователя. Приглашающее сообщение при этом не переводится. Как мы скоро увидим, это должно делаться другим способом.

В дополнение к системной конфигурации, в целях тестирования нужно иметь возможность передать код языка как аргумент командной строки. Для этого изменим конструктор `BookOfTheDayForm`, чтобы можно было передать ему строку культуры и переключить культуру в соответствии со значением этой строки. Создадим экземпляр `CultureInfo` для передачи его свойствам `CurrentCulture` и `CurrentUICulture` текущего потока. Напомним, что `CurrentCulture` используется для форматирования, а `CurrentUICulture` — для загрузки ресурсов:

```
public BookOfTheDayForm(string culture)
{
  if (culture != "")
  {
     CultureInfo ci = new CultureInfo(culture);
     // установить культуру для форматирования
     Thread.CurrentThread.CurrentCulture = ci;
     // установить культуру для ресурсов
     Thread.CurrentThread.CurrentUICulture = ci;
  }
  WelcomeMessage();
  InitializeComponent();
  SetDateAndNumber();
}
```

Экземпляр `BookOfTheDayForm` создается в методе `Main`, который находится в файле `Program.cs`. В этом методе строка, определяющая культуру, передается конструктору `BookOfTheDayForm`:

```
[STAThread]
static void Main(string[] args)
{
  string culture = "";
  if (args.Length == 1)
  {
       culture = args[0];
  }
  Application.EnableVisualStyles();
  Application.Run(new BookOfTheDayForm(culture));
}
```

Теперь можно запускать приложение с использованием опций командной строки. Запустив его, можно видеть в действии форматирующие опции и ресурсы, сгенерированные конструктором Windows Forms. На рисунках 17.16 и 17.17 показаны две локализации при запуске приложения с опциями командной строки `de-DE` и `fr-FR`.

Однако остается проблема с приветствием в окне сообщений. Здесь строки жестко закодированы в исходном тексте программы.


*Рис. 17.16. Форма для выбранного немецкого языка*

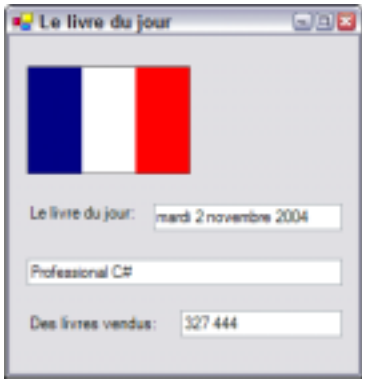


*Рис. 17.17. Форма для выбранного французского языка*

Поскольку эти строки не являются свойствами или элементами внутри формы, конструктор Forms Designer не может извлечь XML-ресурсы, как он делает это со свойствами элементов управления Windows при изменении свойства формы `Localizable`. Придется менять код самостоятельно.

## Использование настраиваемых ресурсов сообщений

Необходимо подготовить переводы жестко закодированного приглашающего сообщения. В табл. 17.5 показаны переводы на немецкий и французский языки.

**Таблица 17.5. Перевод на немецкий и французский язык**

| Имя | Английский | Немецкий | Французский |
|---|---|---|---|
| Good Morning | Good Morning | Guten Morgen | Bonjour |
| Good Afternoon | Good Afternoon | Guten Tag | Bonjour |
| Good Evening | Good Evening | Guten Abend | Bonjour |
| Message1 | This is a localization sample. | Das ist ein Beispiel mit Lokalisierung. | C'est un exemple avec la localisation. |

Исходный код метода `WelcomeMessage()` также должен быть изменен для использования ресурсов. Для строго типизированных ресурсов нет необходимости создавать экземпляр класса `ResourceManager`. Вместо этого можно воспользоваться свойствами строго типизированных ресурсов:

```
public void WelcomeMessage()
{
   DateTime now = DateTime.Now;
   string message;
   if (now.Hour <= 12)
   {
     message = Properties.Resources.Good_Morning;
   }
   else if (now.Hour <= 19)
   {
     message = Properties.Resources.Good_Afternoon;
   }
   else
   {
     message = Properties.Resources.Good_Evening;
   }
   MessageBox.Show(message + "\n" +
     Properties.Resources.Message1);
}
```

В результате запуска программы с указанием английского, немецкого или французского языков мы получим окна сообщений, показанные, соответственно, на рис. 17.18, 17.19 и 17.20.

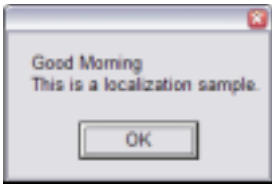


*Рис. 17.18. Окно сообщения на английском языке*

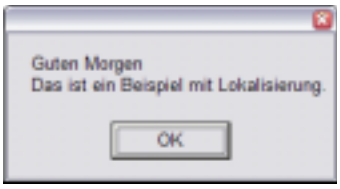


*Рис. 17.19. Окно сообщения на немецком языке*

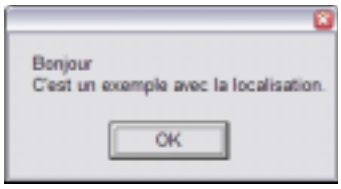


*Рис. 17.20. Окно сообщения на французском языке*

## Автоматическое восстановление ресурсов

Для французской и немецкой версий примера все ресурсы находятся внутри подчиненных сборок. Если изменяются значения не всех меток и текстовых полей, это вовсе не составляет проблемы. В подчиненных сборках должны находиться только изменяемые значения, а все прочие будут извлечены из родительской сборки. Например, для de-AT (Австрия) вы можете изменить значение ресурса *Good Afternoon* на *GrьЯ Gott*, оставив все прочие неизменными. Во время выполнения, когда понадобится найти значение ресурса *Good Morning*, которого нет в подчиненной сборке de-at, поиск продолжится в родительской сборке. Родителем для de-at является de. В случае, если и в сборке de не будет найден нужный ресурс, поиск продолжится в родительской сборке de, а именно — в нейтральной сборке. Нейтральная сборка не имеет кода культуры.

> Следует иметь в виду, что в коде культуры главной сборки нельзя определять никакой культуры!

## Удаленные переводы

Используя ресурсные файлы, очень легко организовать удаленный перевод приложений. Нет необходимости инсталлировать Visual Studio для перевода ресурсных файлов; достаточно простого редактора XML. Недостатком применения редактора XML является то, что он не дает шансов изменить расположение и размер элементов Windows Forms, если переведенный текст не умещается в исходные границы метки или кнопки. Использование конструктор Windows Forms для выполнения перевода — более оптимальный выбор.

Однако Microsoft поставляет в составе .NET Framework SDK инструмент, который отвечает всем этим требованиям: Windows Resource Localization Editor (редактор локализации ресурсов Windows) — `winres.exe` (рис. 17.21). Пользователь этой программы вообще не нуждается в доступе к исходным файлам C#; для перевода необходимы только двоичные или XML-файлы. После завершения перевода можно импортировать ресурсные файлы в проект Visual Studio и построить подчиненные сборки.

Если вы не хотите позволять своему бюро переводов изменять местоположения и размеры меток и кнопок, и они не могут работать с XML-файлами, можете предоставить им простой текстовый файл. Утилитой командной строки `resgen.exe` можно создать из XML текстовый файл:

```
resgen myresource.resX myresource.txt
```

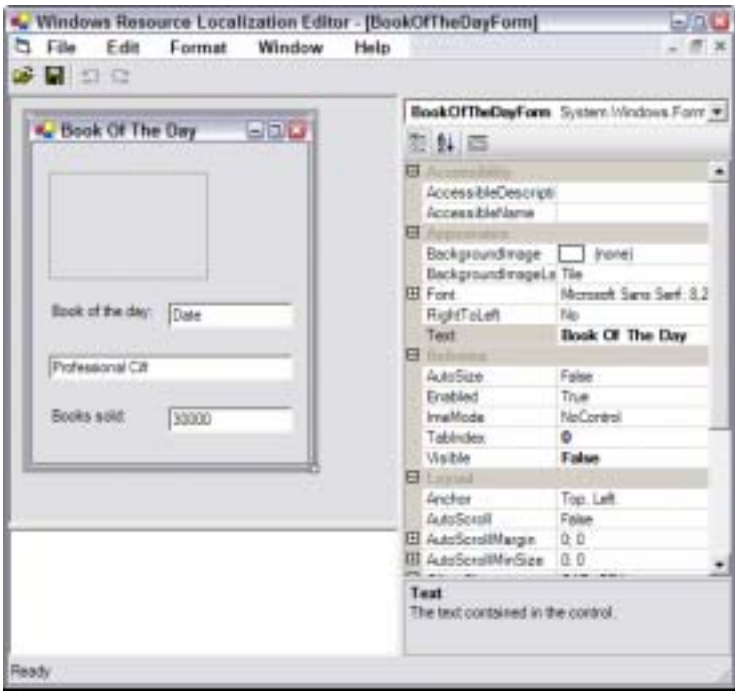


*Рис. 17.21. Редактор Windows Resource Localization Editor*

И после получение готового результата из бюро переводов можно вновь создать из текстового файла XML:

```
resgen myresource.es.txt myresource.es.resX
```

# Локализация в ASP.NET

Локализация приложений ASP.NET выполняется аналогично тому, как это делается с приложениями Windows. В главе 26 обсуждается функциональность приложений ASP.NET; здесь же мы поговорим о том, что касается их локализации. ASP.NET 2.0 и Visual Studio 2005 включают множество новых средств для поддержки локализации. Базовые концепции локализации и глобализации здесь такие же, что были описаны выше. Однако есть несколько специфических моментов, относящихся к ASP.NET.

Как вы уже знаете, в ASP.NET необходимо различать культуру пользовательского интерфейса и культуру, используемую для форматирования. Обе эти культуры могут быть определены как на уровне Web и страницы, так и программно.

Чтобы обеспечить независимость от операционной системы Web-сервера, культура форматирования и культура пользовательского интерфейса могут быть определены элементом `<globalization>` в конфигурационном файле `web.config`:

```
<configuration>
    <system.web>
      <globalization culture="en-US" uiCulture="en-US" />
    </system.web>
</configuration>
```

Если конфигурационный файл должен отличаться для отдельных Web-страниц, то директива `Page` позволяет назначить культуру:

```
<%Page Language="C#" Culture="en-US" UICulture="en-US" %>
```

Если язык страницы должен меняться в зависимости от настроек языка на клиенте, то культура потока может быть установлена программно в соответствии с настройками языка, полученными от клиента. ASP.NET 2.0 предусматривает автоматические установки, которые это делают. Установка культуры в значение `Auto` позволяет автоматически задавать культуру потока в зависимости от клиентских настроек.

```
<%Page Language="C#" Culture="Auto" UICulture="Auto" %>
```

Имея дело с ресурсами, ASP.NET различает ресурсы, которые применяются для всего Web-сайта, и ресурсы, необходимые отдельной странице.

Если ресурсы используются внутри страницы, можно создавать ресурсы страниц, выбирая в меню Visual Studio 2005, при активном конструкторе, команду Tools⇨ Generate Local Resource (Сервис⇨Сгенерировать локальный ресурс). Таким образом, создается подкаталог `App_LocalResources`, в котором сохраняются ресурсные файлы для каждой страницы. Эти ресурсы могут быть локализованы аналогично ресурсам приложений Windows. Ассоциация между элементом управления Web и файлами локальных ресурсов устанавливается атрибутом `meta:resourcekey`, как показано ниже для элемента ASP.NET `Label`. `LabelResource1` — имя ресурса, который может быть изменен в локальном ресурсном файле:

```
<asp:Label ID="Label1" Runat="server" Text="Label"
    meta:resourcekey="LabelResource1"></asp:Label>
```

Для ресурсов, которые должны быть разделены между многими страницами, необходимо создать подкаталог `Application_Resources`. В этот подкаталог можно добавлять ресурсные файлы со своими ресурсами, например, `Messages.resx`. Чтобы ассоциировать элементы управления Web с этими ресурсами, можно использовать средство Expressions (Выражения) в редакторе свойств. Щелчок на кнопке Expressions (Выражения) открывает диалоговое окно Expressions (Выражения), показанное на рис. 17.22. Здесь можно выбрать тип выражения Resources (Ресурс), добавить имя класса (который совпадает с именем файла — здесь генерируется файл строго типизированных ресурсов), а также имя ResourceKey (Ключ ресурса), которое представляет имя ресурса.

В файле ASPX после этого можно увидеть ассоциацию с ресурсом за началом синтаксиса выражений `<%$`:

```
<asp:Label ID="Label1" Runat="server"
  Text="<%$ Resources:Messages, String1 %>">
</asp:Label>
```

# Средства чтения пользовательских ресурсов

Средства чтения ресурсов, входящие в .NET Framework 2.0, позволяют считывать ресурсы из ресурсных файлов и подчиненных сборок. Если же вы хотите поместить ресурсы в другое хранилище (например, в базу данных), то для считывания таких ресурсов можно применять специализированные средства чтения.

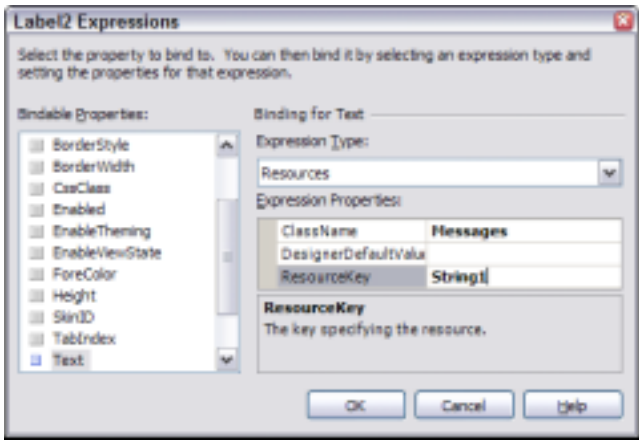


*Рис. 17.22. Диалоговое окно* ***Expressions***

Чтобы использовать средство чтения пользовательских ресурсов, нужно создать набор таких ресурсов и специализированный диспетчер ресурсов. Однако сделать это не трудно, поскольку их классы можно наследовать от существующих классов.

Для примера приложения создадим простую базу данных с единственной таблицей, которая будет хранить сообщения — для каждого поддерживаемого языка в отдельном столбце. В табл. 17.6 представлены столбцы и соответствующие значения.

**Таблица 17.6. Простая база данных с единственной таблицей**

| Ключ | Умолчание | de | es | fr | it |
|---|---|---|---|---|---|
| Welcome | Welcome | Willkommen | Recepción | Bienvenue | Benvenuto |
| Good Morning | Good Morning | Guten Morgen | Buonas díaz | Bonjour | Buona Mattina |
| Good Evening | Good Evening | Guten Abend | Buonas noches | Bonsoir | Buona sera |
| Thank you | Thank you | Danke | Gracias | Merci | Grazie |
| Goodbye | Goodbye | Auf Wiedersehen | Adiós | Au revoir | Arrivederci |

Для средства чтения пользовательских ресурсов создадим библиотеку компонентов с тремя классами: `DatabaseResourceReader`, `DatabaseResourceSet` и `DatabaseResourceManager`.

## Создание `DatabaseResourceReader`

В классе `DatabaseResourceReader` определим два поля: имя источника данных — `dsn`, которое необходимо для доступа к базе данных, и язык, который должен быть возвращен читателем. Эти поля заполняются конструктором класса. Полю `language` присваивается имя культуры, которое передается конструктору с помощью объекта `CultureInfo`:

```
public class DatabaseResourceReader : IResourceReader
{
   private string dsn;
```

```
private string language;
public DatabaseResourceReader(string dsn, CultureInfo culture)
{
  this.dsn = dsn;
  this.language = culture.Name;
}
```

Средство чтения ресурсов должно иметь реализацию интерфейса `IResourceReader`. Этот интерфейс определяет методы `Close()` и `GetEnumerator()`, который возвращает `IDictionaryEnumerator`, в свою очередь, возвращающий ключи и значения ресурсов. В реализации `GetEnumerator()` создадим `HashTable` и поместим туда все ключи и значения для конкретного языка. Далее, можно воспользоваться классом `SqlConnection` из пространства имен `System.Data.SqlClient` для доступа к базе данных SQL Server.

Вызов `Connection.CreateCommand()` создаст объект `SqlCommand`, который применяется для выдачи оператора SQL `SELECT` с целью извлечения данных из базы. Если в качестве языка установлен de, оператор `SELECT` примет вид `SELECT [key], [de] FROM Messages`. Затем с помощью объекта `SqlDataReader` прочитаем все значения из базы и поместим их в `HashTable`. И, наконец, возвратим перечислитель этой `HashTable`.

*Более подробную информацию о доступе к данным из ASP.NET можно найти в главе 19.*

```
public System.Collections.IDictionaryEnumerator GetEnumerator()
{
  Hashtable dict = new Hashtable();
  SqlConnection connection = new SqlConnection(dsn);
  SqlCommand command = connection.CreateCommand();
  if (language == "")
     language = "Default";
  command.CommandText = "SELECT [key], [" + language + "] " +
                        "FROM Messages";
  try
  {
     connection.Open();
     SqlDataReader reader = command.ExecuteReader();
     while (reader.Read())
     {
       if (reader.GetValue(1) != System.DBNull.Value)
            dict.Add(reader.GetString(0), reader.GetString(1));
     }
     reader.Close();
  }
  catch // игнорировать несуществующие столбцы в базе данных
  {
  }
  finally
  {
     connection.Close();
  }
  return dict.GetEnumerator();
}
public void Close()
{
}
```

Поскольку интерфейс IResourceReader наследуется от IEnumerable и IDisposable, также должен быть реализован метод GetEnumerator(), возвращающий интерфейс IEnumerator, и метод Dispose():

```
    IEnumerator IEnumerable.GetEnumerator()
    {
       return this.GetEnumerator();
    }
    void IDisposable.Dispose()
    {
    }
}
```

## Создание DatabaseResourceSet

Класс DatabaseResourceSet может использовать почти все реализации базового класса ResourceSet.

Нам нужен только отдельный конструктор, который инициализирует базовый класс нашим собственным средством чтения ресурса — DatabaseResourceReader. Конструктор ResourceSet позволяет передавать ему объект, реализующий IResourceReader; этому требованию вполне отвечает DatabaseResourceReader:

```
public class DatabaseResourceSet : ResourceSet
{
  internal DatabaseResourceSet(string dsn, CultureInfo culture)
     : base(new DatabaseResourceReader(dsn, culture))
  {
  }
  public override Type GetDefaultReader()
  {
     return typeof(DatabaseResourceReader);
  }
}
```

## Создание DatabaseResourceManager

Третий класс, который нам нужно создать — это пользовательский диспетчер ресурсов. DatabaseResourceManager наследуется от ResourceManager, и для него потребуется только реализовать новый конструктор и переопределить метод InternalGetResourceSet(). В конструкторе создадим новую Hashtable для хранения всех запрашиваемых наборов ресурсов, и присвоим ее полю ResourceSets, определенному в базовом классе:

```
public class DatabaseResourceManager : ResourceManager
{
  private string dsn;
  public DatabaseResourceManager(string dsn)
  {
     this.dsn = dsn;
     ResourceSets = new Hashtable();
  }
```

Методы класса ResourceManager, используемые для обращения к ресурсам (такие как GetString() и GetObject()), вызывают метод InternalGetResourceSet() для доступа к набору ресурсов, откуда может быть возвращено соответствующее значение.

В реализации `InternalGetResourceSet()` сначала проверим, есть ли в хеш-таблице набор ресурсов для запрошенных культуры и ресурса; если он уже есть, возвратим его. Если же набор ресурсов не доступен, создаем новый объект `DatabaseResourceSet` с запрошенной культурой, добавим в хеш-таблицу и вернем вызвавшему коду:

```
protected override ResourceSet InternalGetResourceSet(
    CultureInfo culture, bool createIfNotExists, bool tryParents)
{
  DatabaseResourceSet rs = null;
  if (ResourceSets.Contains(culture.Name))
  {
     rs = ResourceSets[culture.Name] as DatabaseResourceSet;
  }
  else
  {
     rs = new DatabaseResourceSet(dsn, culture);
     ResourceSets.Add(culture.Name, rs);
  }
  return rs;
 }
}
```

## Клиентское приложение для `DatabaseResourceReader`

Использование класса `DatabaseResourceManager` из клиентского приложения не слишком отличается от предыдущих применений класса `ResourceManager`. Единственное отличие состоит в том, что пользовательский класс `DatabaseResourceManager` используется вместо `ResourceManager`. Показанный ниже фрагмент кода демонстрирует применение пользовательского диспетчера ресурсов.

Новый объект `DatabaseResourceManager` создается передачей конструктору строки подключения к базе данных. Затем можно вызывать метод `GetString()`, реализованный в базовом классе, как мы это делали и раньше, передав ключ и необязательный объект `CultureInfo` для указания культуры. Но на этот раз мы получаем ресурс из базы данных, потому что этот диспетчер ресурсов использует классы `DatabaseResourceSet` и `DatabaseResourceReader`.

```
DatabaseResourceManager rm = new DatabaseResourceManager(
"server=localhost;database=LocalizationDemo;trusted_connection=true");
string spanishWelcome = rm.GetString("Welcome",
                        new CultureInfo("es-ES"));
string italianThankyou = rm.GetString("Thank you",
                        new CultureInfo("it"));
string threadDefaultGoodMorning = rm.GetString("Good Morning");
```

# Создание пользовательской культуры

.NET 2.0 имеет новую возможность создания пользовательских культур. Они могут быть созданы для описания культур, которые не предусмотрены в .NET Framework. Некоторые примеры создания пользовательских культур могут быть удобны для поддержки меньшинств в пределах региона, или же для создания субкультур различных диалектов.

Пользовательские культуры и регионы можно создавать с помощью класса `CultureAndRegionInfoBuilder` пространства имен `System.Globalization`. Этот класс находится в сборке `sysglobl`, в файле `sysglobl.dll`.

В следующем примере будет создана новая культура для региона внутри Австрии: Styria (Штирия). Новая культура основана на de-AT и регионе AT. В конструкторе `CultureAndRegionInfoBuilder` новой культуре присваивается имя de-AT-ST. Последним аргументом конструктора также может быть присвоен префикс перечислением `CulturePrefix`. Здесь префикс не используется, поэтому передается значение перечисления `CulturePrefix.None`. После определения имени новой культуры загружаются ее настройки по умолчанию. Метод `LoadDataFromCulture()` загружает все установки культуры de-AT, а метод `LoadDataFromRegion()` — все настройки региона AT.

После создания экземпляра объекта `CultureAndRegionInfoBuilder` можно изменить некоторые аспекты поведения новой культуры, установив значения свойств. Вызов метода `Register()` регистрирует новую культуру в операционной системе. На самом деле, после этого можно найти файл, описывающий новую культуру в каталоге `<windows>\Globalization`. Ищите файлы с расширением `nlp`.

```
// Создание культуры Styria
CultureAndRegionInfoBuilder styria = new CultureAndRegionInfoBuilder(
    "de-AT-ST", CultureAndRegionModifiers.None);
styria.LoadDataFromCulture(new CultureInfo("de-AT"));
styria.LoadDataFromRegion(new RegionInfo("AT"));
styria.Register();
```

Вновь созданная культура теперь может использоваться подобно любой другой:

```
CultureInfo ci = new CultureInfo("de-AT-ST");
Thread.CurrentThread.CurrentCulture = ci;
Thread.CurrentThread.CurrentUICulture = ci;
```

# Резюме

В этой главе описана локализация и глобализация приложений .NET.

В контексте глобализации приложений мы изучили использование пространства имен `System.Globalization` для форматирования чисел и дат, независимых от культуры. Более того, мы узнали, что сортировка строк по умолчанию зависит от культуры, и использовали инвариантную культуру для сортировки, независимой от культуры.

Локализация приложений сопровождается использованием ресурсов. Ресурсы могут быть упакованы в файлы, подчиненные сборки либо пользовательские хранилища, такие как базы данных. Классы, применяемые для локализации, находятся в пространстве имен `System.Resources`.